DOCUMENT RESUME

ED 057 581 EM 009 421

AUTHOR	Lukas, George; And Others
TITLE	LOGO Teaching Sequences on Strategy in Problem-Solving and Story Problems in Algebra. Teacher's Text and Problems.
INSTITUTION	Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	R-2165
PUB DATE	30 Jun 71
NOTE	226p.; Programming-Language as a Conceptual Framework for Teaching Mathematics, Volume Three; See also EM009 419, EM 009 420, EM 009 422
EDRS PRICE	MF-$0.65 HC-$9.87
DESCRIPTORS	*Computer Assisted Instruction; *Computer Programs; *Mathematics Instruction; *Problem Solving; Secondary School Mathematics
IDENTIFIERS	Project LOGO

ABSTRACT

In order to provide high school students with general problem-solving skills, two LOGO computer-assisted instruction units were developed--one on the methods and strategies for solution and a second on the relation between formal and informal representations of problems. In both cases specific problem contexts were used to give definition and articulation to central notions like problem, problem form, solution method, and optimal strategy. The unit on strategies in problem solving illustrates strategy formation in two contexts--extrapolating number sequences and exploring mazes. The unit on story problems in algebra attempts to help students learn to convert a story problem into formal mathematical terms. For more information about the LOGO project, see volumes I, II, and IV of the report (EM 009 419, EM 009 420, and EM 009 422). (JY)

ED057581

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 2165

Volume 3

PROGRAMMING-LANGUAGES AS A CONCEPTUAL FRAMEWORK FOR TEACHING MATHEMATICS

LOGO Teaching Sequences on
Strategy in Problem-Solving
and
Story Problems in Algebra

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, N.W.
Washington, D. C. 20550

Contract NSF-C 615

30 June 1971

EM009 421

ED057581

Report No. 2165, Volume 3

Strategy in Problem-Solving
and
Story Problems in Algebra

FOREWORD

High-school students seldom acquire *general* problem-solving skills. They lack a meta-language for discussing problem-solving and they lack suitable models where the methods of problem-solving are brought to the fore. The two LOGO units in this volume deal directly with these issues. In the first unit, the focus is on methods and strategies for solution; in the second, on the relation between formal and informal representations of problems. In both cases specific problem contexts are used to give definiteness and articulation to central notions like problem, problem form, solution method, and optimal strategy.

In the unit "Strategy in Problem-Solving," we develop extended sequences illustrating strategy formation in two rather different contexts -- extrapolating number sequences and exploring mazes. These examples are designed to give a close-up view into problem-solving work, while showing the flavour of different approaches, and indicating the scope of results possible. Substantive mathematical issues arise naturally along the way and are treated from an intuitive, constructive point of view. We chose our examples to facilitate such treatment -- many other good choices can be made, including studying abstract structures such as groups, games of strategy, and formal manipulation methods.

The unit "Story Problems in Algebra" deals with a central and difficult issue to beginning students. All too often, a student does not realize that the problem of converting a story problem into a problem into formal mathematical terms is a problem in translation. Although the translation rules cannot be made as precise as those for translation between formal systems, they are considerably more precise than those governing the translation say, between English and Chinese. The student who does realize that he is faced with a translation problem still has the problem of formulating the rules -- they are never given to him. In this unit we shall try to give him such "rule-formulating" abilities.

We start with a relatively formal translation problem -- the translation from infix to prefix notation. This serves to shed considerable light on the operations needed to carry forward the less-formal translation and seems a great deal easier. Next we deal with story problems themselves, but instead of taking the usual route of translating from story problems into equations, we translate in the other direction. This route is considerably easier and focuses attention on the translation process itself. For these purposes we develop two kinds of story problem translations -- prefix to story and infix to story. The former is used with a variety of different story problem types involving a single unknown; the latter with problems involving a system of equations.

Volume 3, Part 1

STRATEGY IN PROBLEM-SOLVING

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

George Lukas
Philip Faflick
Wallace Feurzeig
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

Page

1. Number Guessing 1
    1.1 Directing Successive Guesses 4
    1.2 Binary Search 7
2. Number-Sequence Extrapolation 12
    2.1 Sequences and the Extrapolation Problem 12
    2.2 Designing a Sequence Extrapolation Program . . 13
    2.3 Representing Sequences and Generation Rules . . 14
    2.4 Testing Successor Procedures on Sequences . . . 16
    2.5 Enlarging the Procedure Bank 21
    2.6 Automatic Generalization of Successor Procedures 28
    2.7 Extrapolation Using Generalized Successor Procedures 37
    2.8 Transforming Sequences 42
3. Mazes 50
    3.1 Introduction - A Minimal History of Mazes . . . 50
    3.2 Setting up Mazes 51
    3.3 Moving a Mouse Through a Maze 57
    3.4 Relative and Absolute Motion 62
    3.5 New Strategies 71
    3.6 Marking the Path 77
    3.7 History-dependent Strategies 81

Problems

LOGO UNIT ON STRATEGY IN PROBLEM-SOLVING

0. Introduction

This unit develops the art of solving problems by carrying through some examples. We have chosen problem situations that are amenable to different kinds and levels of attack, and that can be approached initially by elementary, intuitively-grasped methods such as guessing and searching. These basic methods are naturally expressed as LOGO procedures. Using these, problem-solving strategies are developed as higher-level procedures either by extending the basic ones or by transforming the problems themselves.

We have sought throughout the LOGO teaching units to counter the harmful impression often given to students that "right" or "best" theorems, algorithms, or solutions are born whole, or are obvious, trivial constructions (or even always exist). In this unit we do not produce optional strategies as instant solutions, like rabbits out of a hat. Instead, the concepts of "good" and "best" strategies arise gradually along with the development of criteria for comparing strategies. Primary emphasis is given to the process of building, testing, and improving strategies.

The unit is divided into three parts. We begin studying strategies for a simple game -- guessing an unknown number. Random and systematic guessing and searching methods are easily introduced in this context.

We next extend number-guessing to a richer problem-solving situation, extrapolating number sequences. Here there is enough additional structure to permit developing a variety of different strategies, some building on others and some essentially independent of others. Some strategies have to do with transforming the sequences to be extrapolated, and others with modifying the rules (LOGO procedures) for generating solutions. The development is carried forward in a "learning" framework. The capabilities of the extrapolation program are continually improved by the incorporation of new strategies. Issues of program organization such as the order of application of strategies, arise naturally.

As a last example, we study an apparently very different situation -- finding effective methods of traversing mazes. The context here is geometric and topological rather than numerical. The basic procedures for moving to adjacent cells are developed first. Using these, various systematic strategies such as "always bear to the left" are defined. Random strategies (and substrategies) are introduced to try to avoid endless repetitive loops and "dead-ends". The possibilities for developing more complex and interesting strategies are further enhanced by marking cells as they are traversed, thus enabling the use of history-dependent strategies.

The LOGO treatment of maze strategies was designed by George Lukas with major contributions from Philip Faflick. George Lukas and Wallace Feurzeig collaborated in the original design of the LOGO number sequence extrapolation material. Philip Faflick contributed to the realization and did the major writing of both this and the maze material. Wallace Feurzeig wrote the introductory section on number-guessing strategies. Pearl Stockwell provided valuable editorial assistance, technical as well as clerical, in the course of transcribing this unit to typed form.

1. NUMBER GUESSING

There are many games of the kind where one person attempts to guess another's "secret" object, whether it is "in this room" or "out of this world", or neither. Among these, Twenty Questions is probably the best known. Some of the number-guessing games are especially well-suited for developing efficient guessing strategies, starting from the simplest guessing procedures. Perhaps the least-sophisticated such game is number guessing without responsive feedback. The situation is: one person thinks of a number, say between 0 and 999, and another tries to guess the number; each time a guess is made the guesser is informed whether or not his guess is correct.

In this form of guessing game no further information about the guess is given -- not even hints like "you're getting warmer" or "you're ice cold". So it's not immediately obvious how we should proceed in making good guesses. Let's start very systematically, by counting up from 0 until we get to the secret number. This is a tedious process, so we'll write a LOGO procedure COUNT-UP to do it for us.

```
TO COUNT-UP /N/                        (/N/ is the current guess)
10 PRINT SENTENCE OF SENTENCE OF
   "IS YOUR NUMBER" /N/ "?"            (Announce the guess)
20 MAKE
    NAME:  "ANSWER"                    (Find whether or not the
    THING:  REQUEST                     guess is correct)
30 TEST IS /ANSWER/ "YES"
40 IF FALSE COUNT-UP                   (If not, increase guess and
   (SUM OF /N/ AND 1)                   try again)
50 IF TRUE PRINT "HOORAY--I GOT IT!"
END
```

Let's use COUNT-UP to guess a number.

```
←COUNT-UP 0                       (COUNT-UP starts by guessing 0)
IS YOUR NUMBER 0?
:NO
IS YOUR NUMBER 1?
:NO
IS YOUR NUMBER 2?
:NO
 :  :  :
 :  :  :

IS YOUR NUMBER 333?
:YES                              (Assuming 333 is the answer)
HOORAY--I GOT IT!
←
```

COUNT-UP takes a long time to get to large numbers. Similarly, the opposite kind of counting procedure, COUNT-DOWN, (for counting backwards from the biggest number) is slow to arrive at small numbers. A more effective guessing procedure should somehow be less biased in favor of high or low numbers. We would like a guessing procedure which better mixes successive guesses among large and small numbers. At the same time, we don't want it to a priori favor evens or primes or, generally, numbers with other special properties. A good way of making such unbiased guesses is by a random or "blind" process in which each possible number in the guessing interval is equally likely to be chosen and where successive choices are independent or uncorrelated. It's easy to write a LOGO procedure BLIND-GUESS to do this kind of guessing.

```
TO BLIND-GUESS
10 MAKE
    NAME:  "GUESS"                  (Make a 3-digit random number
    THING:  RANDOM-NUMBER 3          for use as the guess)
20 PRINT SENTENCE OF SENTENCE OF
   "IS YOUR NUMBER" /GUESS/ "?"     (Announce the guess)
30 MAKE "ANSWER" REQUEST            (Find whether or not the guess
                                     is correct)
40 TEST IS /ANSWER/ "YES"
50 IF FALSE BLIND-GUESS             (If not, guess again)
60 IF TRUE PRINT "HOORAY--I GOT IT!"
END
```

Here is an illustration of the operation of BLIND-GUESS.

```
←BLIND-GUESS
IS YOUR NUMBER 17?
:NO
IS YOUR NUMBER 694?
:NO
IS YOUR NUMBER 17?
:NO, AS I ALREADY SAID!
 : : :
 : : :

IS YOUR NUMBER 333?
:YES
HOORAY--I GOT IT!
←
```

BLIND-GUESS uses the subprocedure RANDOM-NUMBER /L/ which outputs a random number of /L/ digits length.

```
TO RANDOM-NUMBER /L/
10 TEST IS /L/ 0                      (If no more digits are needed)
20 IF TRUE OUTPUT /EMPTY/             (Terminate process)
30 OUTPUT WORD OF                     (Otherwise, attack another
    RANDOM                             random digit and repeat
    RANDOM-NUMBER (DIFF /L/ AND 1) /L/-1 more times)
END
```

Thus:

```
←PRINT RANDOM-NUMBER 2
49
←PRINT RANDOM-NUMBER 10
0924366015
```

An evident way of improving BLIND-GUESS is to insure that it generates a new guess each time, by remembering previous guesses and disallowing repetitions of them. But, no further means of reducing guesses is possible. In this very limited kind of number-guessing situation, the information gained on any incorrect guess is not sufficient to eliminate any other number in the domain.

1.1 Directing Successive Guesses

A genuine improvement in the game comes about by giving the guesser one more piece of information about his guess. Instead of merely responding with a "no" to an incorrect guess, as before, we will also say whether the guess is too high or too low. In this somewhat extended game, the strategy of blind guessing is obviously seen to be unintelligent. If the guesser is told that his guess is too high, he clearly should choose a lower number for his next guess. Similarly, a too low guess should cause the guesser to make a bigger next guess. Let's start by writing such a procedure, TRAP, that operates as follows. Since we have to get this process started, and we don't have any feedback yet, let's begin with a randomly chosen guess. If a guess is too big, TRAP will decrement it by one, if too small TRAP will add one to it.

```
TO TRAP /GUESS/                   (/GUESS/ is the starting guess)
10 PRINT SENTENCE OF
    "MY GUESS IS" /GUESS/
20 PRINT "TELL ME--IS THAT HIGH, LOW, OR OK?"
30 MAKE "ANSWER" REQUEST
40 TEST IS /ANSWER/ "HIGH"
50 IF TRUE TRAP (DIFFERENCE OF /GUESS/ AND 1)
60 TEST IS /ANSWER/ "LOW"
70 IF TRUE TRAP (SUM OF /GUESS/ AND 1)
80 TEST IS /ANSWER/ "OK"
90 IF TRUE PRINT "HOORAY--I GOT IT!"
100 IF FALSE TRAP /GUESS/         (Try again if /ANSWER/ is not
END                               "HIGH", "LOW", or "OK")
```

Here is an example of the operation of TRAP. Let's start it with a 3-digit random number, say 422, generated by RANDOM-NUMBER 3. If our secret number happened to be 131, this is what would happen:

```
←TRAP (RANDOM-NUMBER 3)
MY GUESS IS 422
TELL ME--IS THAT HIGH, LOW, OR OK?
:HIGH
MY GUESS IS 421
TELL ME--IS THAT HIGH, LOW, OR OK?
:HIGH
MY GUESS IS 420
TELL ME--IS THAT HIGH, LOW, OR OK?
   : : :
   : : :
MY GUESS IS 131
TELL ME--IS THAT HIGH, LOW, OR OK?
:OK
HOORAY--I GOT IT!
←
```

So, TRAP leads to lengthy (and dull) exchanges. The difficulty with this procedure is that it is too conservative. The only surprise in its operation is at the start when it finds, from its first (random) guess, an upper or lower bound. From that point on, it closes in on the answer inevitably, but very very slowly. It cannot overshoot (or undershoot) but it can take an enormous number of guesses.

A more efficient strategy comes about from making the guess depend upon both an upper bound and a lower bound. If we know that the answer lies between two numbers, we certainly should choose an intermediate number for our guess. If that number turns out to be too high (too low), it defines a new upper (lower) bound and thus we have further reduced the range of numbers for making our next guess. So this kind of procedure, like TRAP, will converge, though it may make many underestimates and overestimates along the way. And, hopefully, it will be a good deal more efficient on the average.

Let's write such a procedure SQUEEZE to reduce the interval of possible answers at each stage of guessing. The interval is bounded on the low side by /BOTTOM/ and on the high side by /TOP/. SQUEEZE uses a subprocedure INSIDE for randomly choosing as its next guess a number within the specified interval.

```
TO SQUEEZE /BOTTOM/ /TOP/
10 MAKE
    NAME:  "GUESS"
    THING:  INSIDE OF /BOTTOM/ AND /TOP/
20 PRINT SENTENCE OF
    "MY GUESS IS" /GUESS/
30 PRINT "AM I HIGH, LOW, OR OK?"
40 MAKE "ANSWER" REQUEST
50 TEST IS /ANSWER/ "HIGH"       (If guess is too large)
60 IF TRUE SQUEEZE /BOTTOM/      (Try again using /GUESS/-1 as a
    (DIFF OF /GUESS/ AND 1)       better upper bound)
70 TEST IS /ANSWER/ "LOW"        (If guess is too small)
80 IF TRUE SQUEEZE (SUM OF       (Try again using /GUESS/+1 as a
    /GUESS/ AND 1) /TOP/          better lower bound)
90 TEST IS /ANSWER/ "OK"
100 IF TRUE PRINT "HOORAY--
     I GOT IT!"
110 IF FALSE SQUEEZE /BOTTOM/ /TOP/
END
```

The subprocedure INSIDE uses the procedure RANDOM-NUMBER /L/, previously defined, to generate a random increment for /BOTTOM/.

```
TO INSIDE /BOTTOM/ /TOP/
10 MAKE "GAP" (DIFFERENCE OF /TOP/ AND /BOTTOM/)
20 MAKE "R" RANDOM-NUMBER OF (COUNT OF /GAP/)
30 TEST GREATERP /GAP/ /R/      (If interval length is greater
                                 than random increment)
40 IF TRUE OUTPUT               (Use it)
    SUM OF /BOTTOM/ AND /R/
50 GO TO LINE 20                (Otherwise, try again)
END
```

Thus,

```
←PRINT INSIDE OF 35 AND 144
47
←
```

Let's try SQUEEZE in the same example we used above with TRAP.

```
←SQUEEZE 0 999
MY GUESS IS 422
AM I HIGH, LOW, OR OK?
∷HIGH
MY GUESS IS 375
AM I HIGH, LOW, OR OK?
∷HIGH
MY GUESS IS 293
AM I HIGH, LOW, OR OK?
∷HIGH
MY GUESS IS 144
AM I HIGH, LOW, OR OK?
∷HIGH
MY GUESS IS 48
AM I HIGH, LOW, OR OK?
∷LOW
MY GUESS IS 77
AM I HIGH, LOW, OR OK?
∷LOW
MY GUESS IS 139
AM I HIGH, LOW, OR OK?
∷HIGH
MY GUESS IS 118
AM I HIGH, LOW, OR OK?
∷LOW
MY GUESS IS 136
AM I HIGH, LOW, OR OK?
∷HIGH
MY GUESS IS 124
AM I HIGH, LOW, OR OK?
∷LOW
MY GUESS IS 129
AM I HIGH, LOW, OR OK?
∷LOW
MY GUESS IS 131
AM I HIGH, LOW, OR OK?
∷OK
HOORAY--I GOT IT!
←
```

1.2 Binary Search

SQUEEZE shows the kind of improved convergence pattern that we expected relative to TRAP. We can speed up the operation of

SQUEEZE further by noting that there is a great deal of buffeting possible when the sequence of random choices made by INSIDE clusters on one or the other side of the interval, causing it to contract rather slowly even when the gap size has become quite small.

Therefore, we will replace the random incrementation procedure used in INSIDE by a procedure which splits the uncertainty evenly between the lower bound and the upper bound. We'll write a new procedure MIDDLE to directly output the mid-point of the gap as a best guess. Let's first replace line 10 of SQUEEZE by:

```
10 MAKE
    NAME:  "GUESS"
    THING:  MIDDLE OF /BOTTOM/ AND /TOP/
```

MIDDLE is, simply:

```
TO MIDDLE /BOTTOM/ /TOP/
10 OUTPUT QUOTIENT OF (SUM OF /BOTTOM/
    AND /TOP/) AND 2
END
```

Thus,

```
←PRINT MIDDLE OF 2 98
50
←PRINT MIDDLE OF 175 1000
587
←PRINT MIDDLE OF 44 45
44
```

Using MIDDLE and SQUEEZE, let's replay the example used above.

```
←SQUEEZE 0 999
MY GUESS IS 499
AM I HIGH, LOW, OR OK?
::HIGH
MY GUESS IS 249
AM I HIGH, LOW, OR OK?
::HIGH
```

```
MY GUESS IS 124
AM I HIGH, LOW, OR OK?
*LOW
MY GUESS IS 186
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 155
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 139
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 131
AM I HIGH, LOW, OR OK?
*OK
HOORAY--I GOT IT!
←
```

The strategy of splitting the interval of search in half on each round is called binary search. In the example shown, it required fewer guesses than the previously used strategies. Let's see whether or not that was an accidental result of our choice of the interval size or the number to be guessed. (One might observe, for example, that if the unknown number had been 134 instead of 131, binary search would have taken 10 guesses instead of 7.)

Assume we are guessing in the interval (A,B) and that N the number to be guessed is somewhere in the interval, i.e., A≤N≤B. Using TRAP we only eliminate one possibility with each guess. Using binary search our guess, (B-A)/2, eliminates half the possibilities -- the subsequent guessing interval is either (A,(B-A)/2) or ((B-A)/2,B).

Using SQUEEZE, and randomly guessing between A and B, eliminates only one-third of the possibilities -- the argument is as follows: Let's call our guess G, where A≤G≤B. Assume first that G is too high. If so, TRAP eliminates B-G possibilities.

The probability that G is too high is simply (G-A)/(B-A) (i.e., the relative frequency of high guesses). Thus, the number of possibilities eliminated by a too high choice is

$$\frac{(G-A)}{(B-A)}(B-G).$$

Similarly, when G is too low it eliminates G-A possibilities. And this is true with probability (B-G)/(B-A). Thus, the number of possibilities eliminated by a too low choice is

$$\frac{(B-G)}{(B-A)}(G-A).$$

So altogether, adding these, we have

$$\frac{(G-A)}{(B-A)}(B-G)+\frac{(B-G)}{(B-A)}(G-A) = 2\frac{(G-A)(B-G)}{(B-A)}$$

possibilities eliminated on the average. Integrating over all possible guesses within the interval, we get

$$\frac{2}{(B-A)}\int_A^B (G-A)(B-G)dG = (B-A)/3$$

Finally then, SQUEEZE with random guessing eliminates one-third of the possibilities and hence is less efficient than binary search.

Binary search is clearly a better strategy than the others we have considered. We might ask if there is a *best* strategy for guessing numbers. And we might investigate other strategies that might appear better than binary search. For example, if it is good to divide the interval of search into two parts, might it not be better to divide it into three or more parts instead?

Just as we calculated the effect of binary search in reducing remaining choices, we can calculate it for the case where the guess divides the interval into three parts (ternary search) instead. The probability of a too low guess here is 1/3 and the fraction of cases eliminated thereby is 2/3; the probability of a high guess is 2/3 and the fraction of cases eliminated is 1/3. Thus, at each stage 1/3 x 2/3 + 2/3 x 1/3, or 4/9, of the possibilities are eliminated and this is somewhat poorer than the fifty percent reduction obtained with binary search. Similarly, the n-ary search efficiency is

$$\frac{1}{n} \cdot \frac{n-1}{n} + \frac{n-1}{n} \cdot \frac{1}{n} = \frac{2(n-1)}{n^2} \quad ,$$

so the efficiency gets worse with increasing n.

Instead of studying number-guessing strategies further, we can extend the guessing task somewhat and thereby obtain a richer situation that calls for new kinds of strategies. We want to consider guessing problems with more structure than can be provided by guessing a single isolated number. Instead, we will provide more initial information by giving a *sequence* of numbers satisfying some (unknown) generation rule. The task will be to guess the rule or, rather, to guess the next element of the sequence by finding some generation rule that correctly enumerates the numbers in the sequence thus far.

In this problem situation we can develop, not only more varied strategies, but also a system which applies given strategies, and others derived from them, more and more effectively to new problems. We can build a program which in some very definite sense grows and "learns" as it is used.

2. NUMBER-SEQUENCE EXTRAPOLATION

2.1 Sequences and the Extrapolation Problem

We shall be concerned next with another kind of guessing game -- guessing the next number in a sequence of numbers. An illustrative problem is:

given the sequence 0 1 2 3 4 5
what is the next number?

This kind of problem is not mathematically well-defined because, of course, there are an infinite number of different sequences which contain the above one as a subsequence. One might think to restrict the domain of possible answers considerably by demanding that a rule be prescribed for generating an arbitrary number of successors of the given subsequence. But such is not the case -- in fact, for any given finite subsequence, there are an infinite number of such rules. Thus, in the example given, the intended rule might be - each term is the integer successor of the last, modulo 6. In that case the next term would be 0. But the hapless subject might argue with some vehemence that a far more *natural* generation rule is the ordinary integer successor -- which would prescribe 6 as the next term.

Most people feel they have a good idea of what naturalness means. But, how can we impart the same sense to a computer? To resolve this non-mathematical problem -- of choosing the most natural rules -- we let the participants in our game make up the rules for generating their problem sequences and thereby define their criteria of naturalness operationally, by the choices they actually make. Thus, the computer "learns" their meaning of naturalness in an adaptive, evolutionary manner. Our problem then is to design a computer program which, using rules defined by its users, can build effective and natural guessing strategies.

We wish to solve not only "easy" problems, like that in the above example, but also sequences of moderate difficulty such as

-3 1 13 33 61 97

2.2 Designing a Sequence Extrapolation Program

We wish to design a program for predicting the next term of any sequence given to it as an input. Our object is not to make a "perfect" program that would correctly extrapolate any sequence given to it by a mathematician, but only to do so for most sequences devised by high-school students. A conceptually simple way of proceeding is to gradually build up in memory a very large storehouse of sequences, a sort of "sequence bank", and directly look for a copy of the given sequence there so as to find its possible continuation.

The program starts out with an empty sequence bank. In this initial state it must immediately give up on its first problem and ask for the solution. It then stores the entire sequence, including the new term, as the first entry in its sequence bank. The next time it is given a sequence, it can check whether or not this new sequence is a subsequence of one it already contains. If so, it can hazard a possible extrapolation. If not, or if the extrapolation is incorrect, it asks for the solution and stores this new sequence too.

If we think about implementing this scheme, we find that it has some very serious drawbacks. Using it, we must exclude many of the natural sequences of interest. For example, to store just the *constant* sequences (e.g., 1 1 1 1 1) of length 5 for numbers between 0 and 1000 would require nearly 15,000 digits. By comparison, the universal rule for generating the successor for any constant sequence is very easy to write as a program and this requires no more than a few dozen characters. Furthermore, in addition to the enormous memory requirements for storing sequences, the time required to search through tens of thousands of words of storage becomes prohibitive.

How much more reasonable it would be to store the rule itself, instead of some partial sequence it describes. Not only will this be more efficient -- but we will be able to build more powerful strategies using rules than we could using sequences. Let's, therefore, see what is involved in building an extrapolation program based upon rules.

2.3 Representing Sequences and Generation Rules

Let's consider first how we want to express sequences and generation rules in our programs. Sequences can be represented in LOGO the same way they are ordinarily written, i.e., as LOGO number sentences, for example:

```
"1 2 3 4 5"
"0 0 0 0 0 0 0"
"24 48 96"
"14029 871432 -82 40072"
"4"
```

Rules for prescribing sequences, however, can be of many different forms. One standard form of rule gives for any counting number N, the Nth term of the series. Thus, the sequence

1 2 3 4 5 ...

is described, using this form of "indexing" rule, as "the Nth term is the number N". Another standard form of rule, a kind of successor function, prescribes the *next* term of the sequence given only the previous one. The above sequence can be described by the succession rule "the next term is the sum of the last term and 1". This same rule would apply to -

```
395 396 397 398
1002 1003 1004
-3 -2 -1 0 1
```

whereas each of these would require a different indexing rule than the one given above.

Clearly, we would like to regard all of these sequences as subsequences subsumed under a single rule. So we will use successor functions for our standard way of prescribing and generating sequences. The natural representation for a successor function in LOGO is a single-input LOGO procedure. The input to the procedure is a term in a sequence and the output is the next term in that sequence. Thus, for the sequence 1 2 3 4 5, and the related ones just shown, the LOGO successor procedure can be written:

```
TO ADD-ONE /N/
10 OUTPUT SUM OF /N/ AND 1
END
```

This kind of successor function is not universal -- there are sequences that cannot be described by it. An obvious class of such sequences includes those whose terms are functions of two or more preceding terms; for example, sequences which oscillate as follows.

0 1 2 3 4 5 4 3 2 1 0 1 2 ...

It is not possible to write a one-input successor procedure for this sequence since a term is not always determined by its immediate predecessor. The successor of 2, for example, can be either 3 or 1, depending on whether that 2 is in the increasing or decreasing part of the sequence.

Nevertheless, the successor function is a good choice for a first form of sequence generation rule. This kind of rule is both easy to describe and sufficient for describing many different kinds of sequences. It can, in fact, be used for most sequences students are likely to think of. Let's limit ourselves initially then to integer sequences whose terms can be derived from the previous term alone.

2.4 Testing Successor Procedures on Sequences

We can test whether or not a successor procedure describes a sequence by trying it out on the known terms. If ADD-ONE is truly a successor function for the sequence "1 2 3 4 5", then it must correctly output the second term given the first, the third given the second, etc., for all the terms given, as follows.

```
←PRINT ADD-ONE OF 1
2
←PRINT ADD-ONE OF 2
3
←PRINT ADD-ONE OF 3
4
←PRINT ADD-ONE OF 4
5
←
```

Let's write a procedure TESTER to perform this test of a successor's possible validity. TESTER needs two inputs, a sequence and the name of a successor-procedure. It compares the output of the given procedure with the next term, or each pair of terms, in the sequence until either the successor fails or the sequence is exhausted.

```
TO TESTER /SEQUENCE/ AND /PROCEDURE-NAME/
10 TEST EMPTYP OF BUTFIRST OF   (Have all the pairs of terms
    /SEQUENCE/                   been tested?)
20 IF TRUE OUTPUT "TRUE"        (If so, the test is successful)
30 TEST IS SECOND OF /SEQUENCE/ (Is the successor of the first
    EXECUTE OF /PROCEDURE-NAME/  term identical to the second
    AND FIRST OF /SEQUENCE/      term?)
40 IF FALSE OUTPUT "FALSE"      (If not, the successor does not
                                 describe the sequence)
50 OUTPUT TESTER OF
    (BUTFIRST OF /SEQUENCE/)    (Otherwise go on to the rest
    AND /PROCEDURE-NAME/         of the sequence)
END
```

TESTER uses two subprocedures, SECOND and EXECUTE. SECOND simply outputs the second term of a given sequence.

```
TO SECOND /SEQUENCE/
10 OUTPUT FIRST OF BUTFIRST OF /SEQUENCE/
END
```

EXECUTE is the procedure that actually applied the successor procedure to the terms in the sequence. It uses the DO command to perform the procedure named by /PROCEDURE-NAME/:

```
TO EXECUTE /PROCEDURE/ AND /INPUT/
10 DO SENTENCE OF "OUTPUT" AND SENTENCE
    OF /PROCEDURE/ AND /INPUT/
END
```

e.g.,

```
←EXECUTE "ADD-ONE" "5"
6
←
```

TESTER is used as follows.

```
←PRINT TESTER OF "1 2 3 4 5" AND "ADD-ONE"
TRUE
←PRINT TESTER OF "1 3 5 7 9" AND "ADD-ONE"
FALSE
←
```

Let's introduce another successor procedure now.

```
TO ADD-TWO /N/
10 OUTPUT SUM OF /N/ AND 2
END
```

```
←PRINT TESTER OF "1 3 5 7 9" AND "ADD-TWO"
TRUE
←
```

Given a list of successor procedures stored in a "procedure bank", TESTER can be used to determine which of these procedures describes a given sequence. If one or more of them are successful, we can consider using these for predicting the next term of the sequence.

Assuming we have such a /PROCEDURE-BANK/ (i.e., a sentence of successor-procedure names), the next thing we want is a procedure to apply TESTER with a given sequence on each of the procedures listed in /PROCEDURE-BANK/. This new procedure SCAN-P-LIST will scan the list of procedures, apply each one in succession, and output the name of the first procedure that works. If none of the successors describe the sequence, SCAN-P-LIST will output /EMPTY/.

```
TO SCAN-P-LIST /SEQUENCE/ AND /PROCEDURES/
10 TEST EMPTYP /PROCEDURES/      (Have we exhausted the list of
                                  possible successor-procedures?)
20 IF TRUE OUTPUT /EMPTY/        (If so, we have no effective
                                  procedure)
30 TEST TESTER OF /SEQUENCE/     (Does the first procedure describe
    AND (FIRST OF /PROCEDURES/)   the sequence?)
40 IF TRUE OUTPUT FIRST OF       (If so, make it our candidate)
    /PROCEDURES/
50 OUTPUT SCAN-P-LIST OF         (Otherwise, try the rest of the
    /SEQUENCE/ AND (BUTFIRST OF   list)
    /PROCEDURES/)
END
```

Let's build a /PROCEDURE-BANK/ and try out SCAN-P-LIST:

```
←MAKE "PROCEDURE-BANK" "ADD-ONE ADD-TWO"
←PRINT SCAN-P-LIST OF "1 2 3 4 5" AND /PROCEDURE-BANK/
ADD-ONE
←PRINT SCAN-P-LIST OF "1 3 5 7 9" AND /PROCEDURE-BANK/
ADD-TWO
←PRINT SCAN-P-LIST OF "2 4 8 16 32" AND /PROCEDURE-BANK/
                                 (SCAN-P-LIST outputs /EMPTY/)
←PRINT SCAN-P-LIST OF "1" AND /PROCEDURE-BANK/
ADD-ONE
←
```

The last example points to an obvious limitation in SCAN-P-LIST in its current form. The successor procedure ADD-TWO describes the given sequence as well as does ADD-ONE and, indeed, the extrapolated sequence might be "1 3 5 7 ..." instead of "1 2 3 4 ...". But SCAN-P-LIST has stopped short of any further testing of ADD-ONE that might rule it in or out as the actual solution. The requirement that a successor procedure satisfy TESTER for all terms given in the input sequence is not sufficient; the procedure should also extrapolate a correct next term, as judged by the user. Unless one of the procedures in /PROCEDURE-BANK/ can satisfy both these conditions, SCAN-P-LIST should keep searching until the bank is exhausted.

Once a successor procedure is found that satisfies TESTER on all the given terms, we can easily compute the next term given by the procedure, as follows.

```
MAKE "NEXT-TERM" EXECUTE OF (FIRST OF /PROCEDURES/)
 AND (LAST OF /SEQUENCE/)
```

We must then ask the user whether this is, in fact, the *correct* next term. If it is the number he intends, we regard the sequence as solved.[*] If not, we continue scanning the /PROCEDURE-BANK/ for other possible solutions. Assuming we write a subprocedure, APPROVAL, that elicits the user's acceptance of the tentative next term and outputs TRUE or FALSE accordingly, SCAN-P-LIST now looks like this:

```
TO SCAN-P-LIST /SEQUENCE/ AND /PROCEDURES/
10 TEST EMPTYP /PROCEDURES/
20 IF TRUE OUTPUT /EMPTY/
30 TEST TESTER OF /SEQUENCE/ AND (FIRST OF /PROCEDURES/)
40 IF FALSE OUTPUT SCAN-P-LIST OF /SEQUENCE/ AND
    (BUTFIRST OF /PROCEDURES/)
50 MAKE "NEXT-TERM" EXECUTE OF (FIRST OF /PROCEDURES/)
    AND (LAST OF /SEQUENCE/)
60 TEST APPROVAL OF /NEXT-TERM/
70 IF TRUE OUTPUT (FIRST OF /PROCEDURES/)
80 OUTPUT SCAN-P-LIST OF /SEQUENCE/ AND
    (BUTFIRST OF /PROCEDURES/)
END
```

The subprocedure APPROVAL need only type out the tentative next term and test the user's response:

```
TO APPROVAL /NEXT-TERM/
10 TYPE SENTENCE OF "IS THE NEXT TERM" AND SENTENCE OF
    /NEXT-TERM/ AND "?..."      (Ask for user's judgment of
20 MAKE "ANS" REQUEST            tentative next term)
30 TEST IS /ANS/ "YES"
40 IF FALSE OUTPUT "FALSE"      (Rejection)
50 OUTPUT "TRUE"                (Acceptance)
END
```

---

[*] Of course, he may have another sequence in mind which is identical to this one up to this number of terms. Assume for example that a user has in mind the sequence "1 2 4 16 ..." (each number is 2 raised to the previous number). To exclude "1 2 4 8 ..." (each number is 2 times the previous one) from being declared the solution, he merely has to input the starting sequence "1 2 4", since "1 2" is not sufficient. It is incumbent on the user to give enough terms to "uniquely specify" the solution.

We can now try our modified SCAN-P-LIST on /PROCEDURE-BANK/:

```
←PRINT SCAN-P-LIST OF "1 2 3 4" AND /PROCEDURE-BANK/
IS THE NEXT TERM 5?...YES        (The extrapolation is acceptable)
ADD-ONE                     (So SCAN-P-LIST announces the solution)
←PRINT SCAN-P-LIST OF "1" AND /PROCEDURE-BANK/
IS THE NEXT TERM 2?...NO         (This extrapolation is rejected)
IS THE NEXT TERM 3?...YES        (This one is accepted)
ADD-TWO                          (SCAN-P-LIST gives the solution)
←PRINT SCAN-P-LIST OF "2 4 8 16"
 AND /PROCEDURE-BANK/
                                 (No solution was found)
←
```

2.5 Enlarging the Procedure Bank

The power of this extrapolation program depends entirely on the contents of its /PROCEDURE-BANK/ and thus far we have no means of extending it. A good way to do this is to ask the user to supply us with the successor procedures for any sequence SCAN-P-LIST can't solve. If the procedure he types in does indeed describe his sequence, we can add its name to /PROCEDURE-BANK/. The next time we use the extrapolation program the new procedure will be considered along with the others.

The procedure OCCAM sets up SCAN-P-LIST and requests a successor procedure from the user if SCAN-P-LIST fails.

```
TO OCCAM
10 TYPE "WHAT IS YOUR SEQUENCE?..."
20 MAKE "SEQUENCE" REQUEST
30 TEST EMPTYP OF SCAN-P-LIST OF /SEQUENCE/
   AND /PROCEDURE-BANK/         (Note that OCCAM does not announce
40 IF FALSE STOP                the name of the winning procedure)
50 PRINT "I CAN'T DO THAT ONE.  WRITE A LOGO PROCEDURE THAT
   WILL FIND THE SUCCESSOR OF ANY TERM IN THE SEQUENCE AND
   THEN TYPE 'CONTINUE'"
END
```

CONTINUE is a LOGO procedure that asks for the name of the user's procedure, and then tests this procedure on the sequence (using TESTER). If the procedure correctly describes the sequence, OCCAM adds the procedure to /PROCEDURE-BANK/; otherwise, it executes the procedure and prints out the sequence it actually generates.

```
TO CONTINUE
10 TYPE "WHAT WAS THE NAME OF YOUR PROCEDURE?..."
20 MAKE "PROCEDURE" REQUEST
30 TEST TESTER OF /SEQUENCE/ AND /PROCEDURE/
40 IF TRUE MAKE "PROCEDURE-BANK" (SENTENCE OF
    /PROCEDURE-BANK/ AND /PROCEDURE/)
50 IF FALSE PRINT SENTENCE OF SENTENCE OF
    "ERROR ON YOUR PROCEDURE.  I RAN IT STARTING WITH"
    AND FIRST OF /SEQUENCE/ AND "AND GOT:"
60 IF FALSE RUN /PROCEDURE/ AND (FIRST OF /SEQUENCE/)
END
```

The subprocedure RUN prints the sequence described by the faulty successor procedure indefinitely:

```
TO RUN /PROCEDURE/ AND /FIRST-TERM/
10 PRINT /FIRST-TERM/
20 MAKE "NEXT-TERM" EXECUTE OF /PROCEDURE/
    AND /FIRST-TERM/
30 RUN /PROCEDURE/ AND /NEXT-TERM/
END
```

We now have a sequence extrapolation procedure that can automatically be extended with use. To begin with, it knows how to handle two kinds of sequences:

```
←OCCAM
WHAT IS YOUR SEQUENCE?...7 8 9 10
IS THE NEXT TERM 11?...YES
←OCCAM
WHAT IS YOUR SEQUENCE?...22 24 26 28
IS THE NEXT TERM 30?...YES
←
```

And it will quickly need to incorporate new ones:

```
←OCCAM
WHAT IS YOUR SEQUENCE?...2 4 8 16
I CAN'T DO THAT ONE.  WRITE A LOGO PROCEDURE THAT
WILL FIND THE SUCCESSOR OF ANY TERM IN THE SEQUENCE
AND THEN TYPE 'CONTINUE'
←TO TIMES-TWO /N/
>10 OUTPUT PRODUCT OF /N/ AND 2
>END
TIMES-TWO DEFINED
←CONTINUE
WHAT WAS THE NAME OF YOUR PROCEDURE?...TIMES-TWO
←OCCAM
WHAT IS YOUR SEQUENCE?...5 10 20
IS THE NEXT TERM 40?...YES
←
```

Now that things are beginning to move, let's make some changes to shorten OCCAM's printouts and reduce the user's typing somewhat.

```
←EDIT OCCAM
>10 TYPE "SEQUENCE?..."
>50 PRINT "CAN'T DO THAT ONE.  TELL ME HOW AND THEN
     TYPE 'CONTINUE'"
>END
OCCAM DEFINED
←EDIT CONTINUE
>10 TYPE "PROCEDURE NAME?..."
>END
CONTINUE DEFINED
←ABBREVIATE "CONTINUE" AS "CON"
←
```

And now we'll try a new kind of sequence.

```
←OCCAM
SEQUENCE?...1 4 9 16
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO SQUARES /N/
>10 MAKE "X" SUM OF (SQUARE-ROOT OF /N/) AND 1  (Assuming we've
>20 OUTPUT PRODUCT OF /X/ AND /X/                written a SQUARE-
>END                                             ROOT procedure)
←CON
PROCEDURE NAME?...SQUARES
←
```

```
←OCCAM
SEQUENCE?...25 36 49
IS THE NEXT TERM 64?...YES
←
```

Let's check the other branch of CONTINUE by making a deliberate mistake.

```
←OCCAM
SEQUENCE?...1 8 14 21
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO ADD-SEVEN /N/
>10 OUTPUT SUM OF /N/ AND 7
>END
←CON
PROCEDURE NAME?...ADD-SEVEN
ERROR IN YOUR PROCEDURE.  I RAN IT STARTING WITH 1 AND GOT:
1
8
15
22
29
BREAK                   (The user terminates the computation here)
I WAS AT LINE 10 IN EXECUTE
←
```

Let's also see how OCCAM handles ambiguous sequences.

```
←OCCAM
SEQUENCE?...9 16
IS THE NEXT TERM 25?...NO
IS THE NEXT TERM 23?...YES
←
```

Another interesting kind of sequence, that involves a non-numerical successor function, is added next.

```
←OCCAM
SEQUENCE?...1 11 111 1111
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO GROW-ONE /N/
>10 OUTPUT WORD OF /N/ AND "1"
>END
←
```

Unfortunately, OCCAM can't extend its knowledge of GROW-ONE to get this sequence:

```
←OCCAM
SEQUENCE?...2 22 222
CAN'T DO  [etc.]
←TO GROW-TWO /N/
>10 OUTPUT WORD OF /N/ AND "2"
>END
GROW-TWO DEFINED
←CON
PROCEDURE NAME?...GROW-TWO
←OCCAM
SEQUENCE?...1 12
IS THE NEXT TERM 122?...YES
←
```

We can include sequences whose successive terms get smaller instead of larger.

```
←OCCAM
SEQUENCE?...6 5 4
CAN'T DO ...
←TO DIM-ONE /N/
>10 OUTPUT DIFFERENCE OF /N/ AND 1
>END
DIM-ONE DEFINED
←CON
PROCEDURE NAME?...DIM-ONE
←
```

Also, we can include sequences that are not monotone:

```
←OCCAM
SEQUENCE?...0 1 2 3 0 1 2
CAN'T DO ...
←TO MOD-FOUR /N/
>10 OUTPUT REMAINDER OF (SUM OF
     /N/ AND 1) AND 4
>END
MOD-FOUR DEFINED
←
```

With some effort we can even add the factorial sequence (where the factorial function is defined F(N) = N·F(N-1), N>1; F(1)=1)

```
←OCCAM
SEQUENCE?...1 2 6 24
CAN'T DO ...
←TO NEXT-FACTORIAL /N/
>10 OUTPUT FACT OF /N/ AND 1    (NEXT-FACTORIAL uses the following
>END                             recursive subprocedure, FACT)
NEXT-FACTORIAL DEFINED
←TO FACT /N/ AND /COUNTER/
>10 TEST GREATERP /COUNTER/ AND /N/ (Is /COUNTER/ bigger than /N/?)
>20 IF TRUE OUTPUT /COUNTER/        (If so, the answer is /COUNTER/)
>30 OUTPUT PRODUCT OF /COUNTER/ AND (If not, the answer is
      FACT OF (QUOTIENT OF /N/ AND   /COUNTER/ multiplied by the
      /COUNTER/) AND (SUM OF /COUNTER/ result of the indicated
      AND 1)                           recursion)
>END
FACT DEFINED
←CON
PROCEDURE NAME?...NEXT-FACTORIAL
←OCCAM
PROCEDURE?...6 24
IS THE NEXT TERM 120?...YES
←
```

Our extrapolation program is beginning to acquire some power. But, it has some unnecessary operational flaws. For instance, it can make annoyingly repetitive extrapolations. As an example, look at what happens now if our input sequence is simply "1".

```
IS THE NEXT TERM 2?...NO
IS THE NEXT TERM 3?...NO
IS THE NEXT TERM 2?...NO
IS THE NEXT TERM 4?...NO
IS THE NEXT TERM 8?...NO
IS THE NEXT TERM 11?...NO
IS THE NEXT TERM 12?...NO
IS THE NEXT TERM 0?...NO
IS THE NEXT TERM 2?...NO
```

We see here that 2 was offered as a next term three times. Let's modify OCCAM so that it will never try any /NEXT-TERM/ more than once. All we need do is keep a running list of all the /NEXT-TERM/s that failed, and check to see whether or not a /NEXT-TERM/ is already on the list before we submit it to the user.

This list, call it /BAD-NEXT-TERMS/, should be made empty each time OCCAM is used. One new line does this.

```
←EDIT OCCAM
>5 MAKE "BAD-NEXT-TERMS" /EMPTY/
>END
OCCAM DEFINED
←
```

Related changes must be made to APPROVAL. First of all, APPROVAL should check to see if its input /NEXT-TERM/ is on this list of rejected extrapolations. It uses a subprocedure CONTAINSP, defined below, to do this test.

```
←EDIT APPROVAL
>5 TEST CONTAINSP /BAD-NEXT-TERMS/ /NEXT-TERM/
```

If /NEXT-TERM/ is on the list, we know without further query that it is not the correct next term. Thus:

```
>6 IF TRUE OUTPUT "FALSE"
```

Also, APPROVAL should add any unsuccessful extrapolations to the list as soon as they have been rejected:

```
>35 IF FALSE MAKE "BAD-NEXT-TERMS" (SENTENCE
    OF /BAD-NEXT-TERMS/ AND /NEXT-TERM/)
>END
APPROVAL DEFINED
```

The procedure CONTAINSP is a predicate with two inputs, a list and an element.

```
TO CONTAINSP /LIST/ AND /ELEMENT/
10 TEST EMPTYP /LIST/              (Have we exhausted the list?)
20 IF TRUE OUTPUT "FALSE"          (If so, the list did not contain
                                    the element)
30 TEST IS (FIRST OF /LIST/)       (Is the first thing on /LIST/
   /ELEMENT/                        the element?)
40 IF TRUE OUTPUT "TRUE"           (Yes, /LIST/ contains /ELEMENT/)
50 OUTPUT CONTAINSP OF (BUTFIRST  (Otherwise, try the rest of
   OF /LIST/) AND /ELEMENT/         the list)
END
```

OCCAM's responses to the sequence "1" are now non-repetitive:

```
IS THE NEXT TERM 2?...NO
IS THE NEXT TERM 3?...NO
IS THE NEXT TERM 4?...NO
IS THE NEXT TERM 8?...NO
IS THE NEXT TERM 11?...NO
IS THE NEXT TERM 12?...NO
IS THE NEXT TERM 0?...NO
CAN'T DO ...
```

2.6 Automatic Generalization of Successor Procedures

OCCAM has another more basic limitation. If we look at the current procedure-bank --

```
←PRINT /PROCEDURE-BANK/
ADD-ONE ADD-TWO TIMES-TWO SQUARES ADD-SEVEN GROW-ONE
GROW-TWO DIM-ONE MOD-FOUR NEXT-FACTORIAL
←
```

we find many successor procedures that are virtually identical in form. In particular, the LOGO definitions for the procedures ADD-ONE, ADD-TWO, and ADD-SEVEN differ only by a constant. (The same is true for GROW-ONE and GROW-TWO.) These are essentially the same procedures. Moreover, many more procedures of the same kind will surely be added as OCCAM is used. Clearly it is inefficient to have so many instances of virtually the same generating procedures. More importantly, though, no matter how many instances of the same general form OCCAM has seen, it will nevertheless be unable to extrapolate any new such instance. In effect, it fails to generalize its experience. If OCCAM somehow were able to construct, from a single such instance, a more general procedure which included *all* similar instances, it would gain enormously in power. In particular, it would then be able to extrapolate sequences it has never seen before. There is a simple way of accomplishing this for many procedures of interest.

Let's redefine ADD-ONE as follows.

```
TO ADD-ONE /N/ AND /DUMMY/        (/DUMMY/ is a new input)
10 OUTPUT SUM OF /N/ AND /DUMMY/  (/DUMMY/ replaces the old "1"
END                                 here)
```

The original ADD-ONE describes a particular arithmetic sequence. Replacing the constant "1" with a dummy variable has the effect of generalizing the original one-input successor function to a two-input successor which describes the entire family of arithmetic sequences. If /DUMMY/ is 1, we get the original ADD-ONE again; with /DUMMY/ set to 2, we have ADD-TWO, and so on for all such arithmetic sequences (i.e., those whose successive terms differ by some constant). Similarly, a generalized TIMES-TWO

```
TO TIMES-TWO /N/ AND /DUMMY/
10 OUTPUT PRODUCT OF /N/ AND /DUMMY/
END
```

will describe all *geometric sequences*, sequences whose terms are some constant multiple of their predecessor. Let's show next how OCCAM can construct these generalized procedures. Then we will see how it can use them to improve its extrapolation capabilities. In the examples we looked at, ADD-ONE and TIMES-TWO, the generalization was accomplished by replacing the constant with a variable, /DUMMY/, and appending /DUMMY/ to the title. Some procedures contain two or more constants that could be replaced by variables but, instead of replacing all of these, we will content ourselves with simply replacing the *first* constant that occurs within the procedure.

To change a one-input successor procedure into such a "generalized" two-input form, we will need to be able to write procedures for modifying other procedures. We can do this using the LOGO command DO along with two LOGO operations, LINES and TEXT, which enable us to extract from any procedure the list of line numbers and the associated instructions in specified lines. The LOGO operation LINES takes as its input a procedure name and outputs a sentence made up of the line numbers contained in the procedure, for example:

```
←PRINT LINES OF "APPROVAL"
5 6 10 20 30 35 40 50
←
```

The built-in operation TEXT gives us access to the instruction lines themselves. For example:

```
←PRINT TEXT OF "APPROVAL" AND 50
50 OUTPUT "TRUE"
←
```

With access to the contents of a procedure, we can write a procedure SCAN-LINES which goes through a procedure line-by-line looking for a constant to replace.

```
TO SCAN-LINES /PROCEDURE/ AND /LINES/  (LINES is the sentence of
                                         line numbers)
10 TEST EMPTYP /LINES/        (Have we gone through all the lines?)
20 IF TRUE OUTPUT "FALSE"     (If so, there were no constants to
                               replace)
30 MAKE "LINE-TEXT" TEXT OF     (Get the text of the first line)
    /PROCEDURE/ AND (FIRST
    OF /LINES/)
40 TEST CONTAINS-NUMBER OF      (Are there any numbers in the
    BUTFIRST OF /LINE-TEXT/      line besides the line number?)
50 IF FALSE OUTPUT SCAN-LINES   (If not, check the next line)
    OF /PROCEDURE/ AND BUTFIRST
    OF /LINES/
```

```
60 CHANGE /PROCEDURE/ AND         (If so, edit the procedure)
    /LINE-TEXT/                   (and indicate that we have
70 OUTPUT "TRUE"                   generalized the procedure)
END
```

The subprocedure CONTAINS-NUMBER simply applies NUMBERP to every word (excluding the line number) in the text of an instruction line:

```
TO CONTAINS-NUMBER /LINE/
10 TEST EMPTYP /LINE/             (Have we exhausted the line?)
20 IF TRUE OUTPUT "FALSE"         (If so, it contained no constant)
30 TEST NUMBERP OF FIRST OF /LINE/  (Is first word a number?)
40 IF TRUE OUTPUT "TRUE"          (The answer is yes)
50 OUTPUT CONTAINS-NUMBER OF      (If not, repeat the process with
    (BUTFIRST OF /LINE/)           the rest of the line)
END
```

SCAN-LINES uses CHANGE to edit a procedure known to contain a constant. (The LOGO command DO takes its input as a LOGO instruction and executes it as such.)

```
TO CHANGE /PROCEDURE/ AND /LINE-TEXT/
10 DO SENTENCE OF "EDIT" AND /PROCEDURE/  (Put LOGO into edit mode)
20 DO SENTENCE OF (TEXT OF /PROCEDURE/    (Append /DUMMY/ onto
    AND 0) (AND "/DUMMY/")                 line 0, the title line)
30 DO SENTENCE OF FIRST OF
    /LINE-TEXT/ AND REWRITE OF            (Rewrite the line contain-
    BUTFIRST OF /LINE-TEXT/                ing a number)
40 DO "END"                               (Leave edit mode)
END
```

CHANGE's subprocedure, REWRITE, is very much like CONTAINS-NUMBER; it scans a line looking for a constant. However, REWRITE actually reconstructs the line, changing the first number in it to the word "/DUMMY/".

```
TO REWRITE /LINE/
10 TEST NUMBERP OF FIRST OF     (Is the first word a number?)
    /LINE/
20 IF TRUE OUTPUT SENTENCE OF   (If so, replace it with "/DUMMY/"
    "/DUMMY/" AND BUTFIRST OF    and append the rest of the line)
    /LINE/
30 OUTPUT SENTENCE OF (FIRST OF (Otherwise, put it back in the
    /LINE/) AND (REWRITE OF      line and repeat the process
    BUTFIRST OF /LINE/)          with the rest of the line)
END
```

The outer procedure GENERALIZE simply invokes SCAN-LINES, giving it a procedure name and the associated list of the procedure's line numbers:

```
TO GENERALIZE /PROCEDURE/
10 OUTPUT SCAN-LINES OF /PROCEDURE/ AND
    (LINES OF /PROCEDURE/)
END
```

Let's try out these new generalization capabilities:

```
←PRINT GENERALIZE OF "ADD-ONE"  (ADD-ONE has now been generalized)
TRUE
←LIST ADD-ONE
TO ADD-ONE /N/ /DUMMY/
10 OUTPUT SUM OF /N/ AND /DUMMY/
END
←
```

Not all successor procedures can be generalized in this way. GENERALIZE acknowledges its inability to generalize a procedure with the output "FALSE". For example:

```
←TO SQUARE /N/
10 OUTPUT PRODUCT OF /N/ AND /N/
END
SQUARE DEFINED
←PRINT GENERALIZE OF "SQUARE"
FALSE
←
```

Now that we can construct some generalized procedures, let's see how to incorporate these in our extrapolation program.

SCAN-P-LIST and TESTER, as they stand, will not work for two-input successor procedures. We still need these procedures in their present form to handle the ungeneralizable successors in /PROCEDURE-BANK/, but we will also require precisely parallel procedures to do the same things for generalized successor procedures that we will accumulate in /GENERAL-BANK/.

First, let's write a new TESTER. To determine whether or not a general successor correctly describes a given sequence, we will have to test it on every term across a range of values for /DUMMY/. A reasonable way to do this is to divide the problem into two parts -- to write one procedure, GENERAL-TESTER, to run through the range of /DUMMY/ values, and another, GEN-TEST-2 to check each pair of known terms in the problem-sequence with some fixed value of /DUMMY/. GEN-TEST-2 is almost identical to TESTER.

```
TO GEN-TEST-2 /SEQUENCE/ AND /PROCEDURE/ AND /DUMMY/
10 TEST EMPTYP BUTFIRST OF /SEQUENCE/  (Have we checked all the
                                         pairs of terms?)
20 IF TRUE OUTPUT "TRUE"        (If so, the procedure - with this
                                 /DUMMY/ - is accepted)
30 TEST IS (SECOND OF /SEQUENCE/) (Does the procedure correctly
    (EXECUTE-2 OF /PROCEDURE/      give the second term of the
    AND FIRST OF /SEQUENCE/        sequence?)
    AND /DUMMY/
40 IF FALSE OUTPUT "FALSE"        (If not, it is rejected)
50 OUTPUT GEN-TEST-2 OF BUTFIRST  (If so, repeat the test on
    OF /SEQUENCE/ AND /PROCEDURE/  the rest of the sequence)
    AND /DUMMY/
END
```

The subprocedure EXECUTE-2 is a simple extension of EXECUTE designed to execute two-input procedures:

```
TO EXECUTE-2 /PROCEDURE/ AND /INPUT-1/ AND /INPUT-2/
10 DO SENTENCE OF "OUTPUT" AND SENTENCE OF /PROCEDURE/
    AND SENTENCE OF /INPUT-1/ AND /INPUT-2/
END
```

Now we will have to choose a range of values to use with GEN-TEST-2. Should we try two values? ten? one hundred? a thousand? On the one hand, we want to use a "big" range; the larger the range the more effective the successor procedures will be. But, even if it checks all values from 0 to 1000, a generalized ADD-ONE, for example, would miss the sequence -

1 1002 2003 3004

But a person confronted with such a sequence of large numbers would not carry out this kind of generalization technique for 1000 successive values of a parameter. Rather, he would try to transform the problem into a more tractable form where rules using small parameter values might be effective. Perhaps the most natural such transformation is to replace the original sequence by one consisting of the differences between successive terms of the original -- obviously a good choice for the above sequence. Use of such transformations can vastly improve our extrapolation capabilities. We will develop some procedures of this kind after seeing the kind of benefits we obtain with GENERALIZE.

Since we are not depending upon GENERALIZE to solve problems with large parameters, and since the larger the range chosen the longer it takes to do the checking, we shall choose a small range of values. The range zero through ten is a natural one for people doing parametric calculations so let's make the same choice for our program. GENERAL-TESTER then will test a given generalized procedure on a sequence using values of /DUMMY/ between 0 and 10 successively. It will output the first /DUMMY/ that is successful across all known terms in the sequence. If no /DUMMY/ is successful, it will output /EMPTY/.

```
TO GENERAL-TESTER /SEQUENCE/ AND /PROCEDURE/
10 MAKE "DUMMY" 0
20 TEST GEN-TEST-2 OF /SEQUENCE/ AND /PROCEDURE/ AND /DUMMY/
30 IF TRUE OUTPUT /DUMMY/
40 TEST IS /DUMMY/ 10
50 IF TRUE OUTPUT /EMPTY/
60 MAKE "DUMMY" (SUM OF /DUMMY/ AND 1)
70 GO TO LINE 20
END
```

Let's try GENERAL-TESTER with our generalized ADD-ONE:

```
←PRINT GENERAL-TESTER OF "1 5 9 13 17" AND "ADD-ONE"
4                                (This is the "winning" /DUMMY/)
←PRINT GENERAL-TESTER OF "2 4 8 16" AND "ADD-ONE"
                                 (There is no winning /DUMMY/)
←
```

We will also need a procedure similar to SCAN-P-LIST which uses GENERAL-TESTER to scan a list of generalized successor procedures. The only differences between this procedure, call it SCAN-G-LIST, and the old SCAN-P-LIST are due to the extra input, /DUMMY/; SCAN-G-LIST will have to save the output of GENERAL-TESTER (either a number or /EMPTY/, rather than "TRUE" or "FALSE") and use it in EXECUTE-2 to calculate the next term.

```
TO SCAN-G-LIST /SEQUENCE/ AND /PROCEDURES/
10 TEST EMPTYP /PROCEDURES/      (Have we exhausted the list?)
20 IF TRUE OUTPUT /EMPTY/        (Yes, no winners in the list)
30 MAKE "WINNING-DUMMY" GENERAL-TESTER  (Test the first procedure
    OF /SEQUENCE/ AND (FIRST OF           - save the result)
    /PROCEDURES/)
40 TEST EMPTYP /WINNING-DUMMY/     (Did the procedure fail?)
50 IF TRUE OUTPUT SCAN-G-LIST OF  (If so, try the rest of the
    /SEQUENCE/ AND (BUTFIRST OF     list)
    /PROCEDURES/)
60 MAKE "NEXT-TERM" EXECUTE-2 OF  (If the procedure succeeded,
    FIRST OF /PROCEDURES/ AND       use it to calculate a next
    LAST OF /SEQUENCE/ AND          term)
    /WINNING-DUMMY/
70 TEST APPROVAL OF /NEXT-TERM/   (Is this extrapolation accepted?)
80 IF TRUE OUTPUT FIRST OF /PROCEDURES/  (Yes)
90 OUTPUT SCAN-G-LIST OF /SEQUENCE/  (No, so repeat the process
    AND (BUTFIRST OF /PROCEDURES/)    with the rest of the proce-
END                                   dures)
```

We now have procedures that can generalize successor functions and procedures that can test lists of generalized successors against a given sequence. Linking these procedures to our existing sequence extrapolation is straightforwardly done.

First of all, we need to modify CONTINUE so as to add new procedures to /GENERAL-BANK/. When a new successor satisfies TESTER, CONTINUE should give it to GENERALIZE. If GENERALIZE outputs "TRUE", the generalized procedure can be added to /GENERAL-BANK/, otherwise the procedure is not generalizable and must be appended to our other list, /PROCEDURE-BANK/. With these changes CONTINUE looks as follows.

```
TO CONTINUE
10 TYPE "PROCEDURE NAME?..."
20 MAKE "PROCEDURE" REQUEST
30 TEST TESTER OF /SEQUENCE/ AND /PROCEDURE/
40 IF FALSE PRINT SENTENCE OF "ERROR IN YOUR PROCEDURE. I
    RAN IT STARTING WITH" AND SENTENCE OF (FIRST OF
    /SEQUENCE/) AND "AND GOT:"
50 IF FALSE RUN /PROCEDURE/ AND (FIRST OF /SEQUENCE/)
60 TEST GENERALIZE OF /PROCEDURE/  (Does the procedure generalize?)
70 IF TRUE MAKE "GENERAL-BANK"     (If so, add the generalized
    SENTENCE OF /GENERAL-BANK/      procedure to /GENERAL-BANK/)
    AND /PROCEDURE/
80 IF FALSE MAKE "PROCEDURE-BANK"  (Otherwise, put it with the
    SENTENCE OF /PROCEDURE-BANK/    other one-input successor)
    AND /PROCEDURE/
END
```

All that remains to do is to give OCCAM the option of using generalized procedures (if there are any) for its extrapolations. So we will have to do SCAN-G-LIST after it does SCAN-P-LIST. This is done by adding a single line to OCCAM.

```
35 IF TRUE TEST EMPTYP OF
    SCAN-G-LIST OF /SEQUENCE/   (If SCAN-P-LIST fails,
    AND /GENERAL-BANK/           test SCAN-G-LIST)
```

2.7 Extrapolation Using Generalized Successor Procedures

We're now able to try the new and more powerful extrapolation program we have been building. Since many of the successor functions in /PROCEDURE-BANK/ can be expressed more efficiently with a generalized successor, let's start once again with a completely blank slate.

```
←MAKE "PROCEDURE-BANK" /EMPTY/
←OCCAM
SEQUENCE?...5 5 5 5
CAN'T DO THAT ONE.  TELL ME NOW AND THEN TYPE 'CONTINUE'
←TO FIVERS /N/
>10 OUTPUT 5
>END
FIVERS DEFINED
←CON
PROCEDURE NAME?...FIVERS
←OCCAM
SEQUENCE?...8 8 8 8
IS THE NEXT TERM 8?...YES
←
```

A user who didn't know about our generalizer might be surprised to see OCCAM solving sequences for which it wasn't specifically given a rule. Instead of keeping him completely in the dark about OCCAM's methods though, we might like to let him know what particular procedure(s) it employed to extrapolate his sequence. So let's modify SCAN-P-LIST to declare a "winning" ungeneralized procedure, as follows.

```
65 IF TRUE PRINT SENTENCE OF "THE WINNING PROCEDURE WAS"
    AND (FIRST OF /PROCEDURES/)
```

In modifying SCAN-G-LIST we will want to announce the /WINNING-DUMMY/ as well as the procedure name:

```
75 IF TRUE PRINT SENTENCE OF "THE WINNING PROCEDURE WAS"
    AND SENTENCE OF (FIRST OF /PROCEDURES/) AND SENTENCE
    OF "FOR /DUMMY/ EQUALS" AND /WINNING-DUMMY/
```

Now let's try the last example again:

```
←OCCAM
SEQUENCE?...8 8 8 8
IS THE NEXT TERM 8?...YES
THE WINNING PROCEDURE WAS FIVERS FOR /DUMMY/ EQUALS 8
←LIST FIVERS                        (The original FIVERS procedure
TO FIVERS /N/ /DUMMY/                was generalized as shown)
10 OUTPUT /DUMMY/
END
←
```

The generalized procedure FIVERS can solve all sequences of constant terms (between 0 and 10). We can effect comparable generalizations for arithmetic and geometric sequences as easily.

```
←OCCAM
SEQUENCE?...99 101 103 105
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO ADD-TWO /N/
←10 OUTPUT SUM OF /N/ AND 2
←END
ADD-TWO DEFINED
←CON
PROCEDURE NAME?...ADD-TWO
←OCCAM
SEQUENCE?...99 105 111
IS THE NEXT TERM 117?...YES
THE WINNING PROCEDURE WAS ADD-TWO FOR /DUMMY/ EQUALS 6
←OCCAM
SEQUENCE?...8 16 32
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
  : : :                    (TIMES-TWO is entered here)
←CON
SEQUENCE NAME?...TIMES-TWO
←OCCAM
SEQUENCE?...1 9 81 729
IS THE NEXT TERM 6561?...YES
THE WINNING PROCEDURE WAS TIMES-TWO FOR /DUMMY/ EQUALS 9
←
```

We can also add our old procedures DIM-ONE and GROW-ONE, and these will also be generalized.

Then, for example:

```
←OCCAM
SEQUENCE?...57 576 5766 57666
IS THE NEXT TERM 576666?...YES
THE WINNING PROCEDURE WAS GROW-ONE FOR /DUMMY/ EQUALS 6
←OCCAM
SEQUENCE?...70 61 52 43
IS THE NEXT TERM 34?...YES
THE WINNING PROCEDURE WAS DIM-ONE FOR /DUMMY/ EQUALS 9
←
```

The procedures SQUARES, MOD-FOUR, and NEXT-FACTORIAL can also be generalized in the same way, but the results are less useful. In all these cases, the generalized procedures simply describe subsets of the sequence described by the original, ungeneralized procedure. The original SQUARES, for example, describes the squares of the integers; the generalized SQUARES procedure is:

```
TO SQUARES /N/ /DUMMY/
10 MAKE "X" SUM OF SQUARE-ROOT OF /N/ AND /DUMMY/
20 OUTPUT PRODUCT OF /X/ AND /X/
END
```

Instead of describing sequences which are more general than the original, this procedure merely describes various subsets of the original sequence. Perhaps the most useful case is the subset consisting of the squares of the odd integers:

```
←OCCAM
SEQUENCE?...1 9 25 49
IS THE NEXT TERM 81?...YES
THE WINNING PROCEDURE WAS SQUARES FOR /DUMMY/ EQUALS 2
```

The procedures SQUARES, MOD-FOUR, and NEXT-FACTORIAL can, however, be otherwise defined in ways that lead to very useful generalizations. For example, if we define SQUARES:

```
TO SQUARES /N/
10 OUTPUT UP-ROOT OF 2 AND /N/
END
```

As before, SQUARES computes $(\sqrt{/N/}+1)^2$. It uses the procedure UP-ROOT which, in turn, uses two standard subprocedures, ROOT /M/ /N/ and POWER /M/ /N/, for getting /M/th roots and /M/th powers.

```
TO UP-ROOT /M/ AND /N/
10 MAKE "X" (ROOT OF /M/ AND /N/)  (This gets the /M/th root of /N/)
20 OUTPUT POWER OF /M/ AND         (This raises /X/+1 to the /M/th
    (SUM OF /X/ AND 1)              power)
END
```

When this version of SQUARES is generalized, it becomes:

```
TO SQUARES /N/ AND /DUMMY/
10 OUTPUT UP-ROOT OF /DUMMY/ AND /N/
END
```

And this is a genuinely useful extension -- it effectively generalizes SQUARES to describe sequences of integers raised to various integer powers. For example, if /DUMMY/ is 3, we get the sequence of cubes 1 8 27 64...; if /DUMMY/ is 4 we get the 4th powers 1 16 81 256... and so on.

Similarly, let us rewrite the MOD-FOUR successor as,

```
TO MOD-FOUR /N/
10 TEST IS /N/ 4
20 IF TRUE OUTPUT 0
30 OUTPUT SUM OF /N/ AND 1
END
```

It will then truly generalize as the successor for the family of modulo-/DUMMY/ sequences (except when /N/ is greater than /DUMMY/).

```
TO MOD-FOUR /N/ /DUMMY/
10 TEST IS /N/ /DUMMY/
20 IF TRUE OUTPUT 0
30 OUTPUT SUM OF /N/ AND 1
END
```

Thus, by exercising some care in how the original, ungeneralized procedures are defined, the strategy of generalizing procedures can have a wide scope of application. It can, however, lead to trouble with certain procedures. Consider, for example, what it does with the procedure DIV-TWO:

```
←OCCAM
SEQUENCE?...16 8 4
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO DIV-TWO /N/
>10 OUTPUT QUOTIENT OF /N/ AND 2
>END
DIV-TWO DEFINED
←CON
PROCEDURE NAME?...DIV-TWO
←OCCAM
SEQUENCE?...8 4 2
DIVISION BY ZERO                  (Error comment - illegal operation)
I WAS AT LINE 10 IN DIV-TWO       (LOGO terminates the program)
←
```

The "bug" is easily located. When GENERAL-TESTER tried to test the generalized DIV-TWO procedure with its first /DUMMY/ value, 0, it called DIV-TWO OF 8 AND 0 which attempted to divide 8 by 0. We can fix this by writing DIV-TWO in a tricky way replacing the divisor 2, by SUM OF 1 AND 1. Thus, in the generalized form, /DUMMY/ will not replace the entire divisor, so division by zero will not occur.

```
←TO DIV-TWO /N/
>10 OUTPUT QUOTIENT OF /N/ AND (SUM OF 1 AND 1)
>END
DIV-TWO DEFINED
←CON
```

```
PROCEDURE NAME?...DIV-TWO
←OCCAM
SEQUENCE?...27 9 3
IS THE NEXT TERM 1?...YES
THE WINNING PROCEDURE WAS DIV-TWO FOR /DUMMY/ EQUALS 2
←LIST DIV-TWO
TO DIV-TWO /N/ /DUMMY/
10 OUTPUT QUOTIENT OF /N/ AND (SUM OF /DUMMY/ AND 1)
END
←
```

Situations of this kind, which call for rewriting a procedure to eliminate an illegal operation, are unavoidable but they do not occur frequently.

2.8 Transforming Sequences

Generalization has measurably increased OCCAM's power. Before we introduced GENERALIZE, OCCAM had a procedure-bank containing ten ungeneralized procedures and thus could identify segments of ten infinite sequences. Once again OCCAM has ten procedures in its banks, one ungeneralized and nine in the /GENERAL-BANK/. But each generalized procedure actually describes 11 different successor procedures (corresponding to 11 different values of /DUMMY/). So OCCAM can identify, all together, any segment of 100 different sequences.

We can, of course, considerably expand the number of procedures in OCCAM without changing its structure. Adding new rules will widen its scope at the rate of one sequence for every ungeneralized procedure and 11 (or more if we increase the range of /DUMMY/) for every new entry in /GENERAL-BANK/. We will certainly add new rules, but there is a very different direction for expansion that will be far more effective in boosting OCCAM's power. Up to now we have concentrated our effort on a strategy for generalizing sequence generation rules. Let's look now at another kind of strategy, for transforming the given sequences directly. As we noted before, even testing /DUMMY/ between 0 and 1000, OCCAM would fail to find the next term of the sequence:

```
1 1002 2003 3004 ...
```

Subtracting every term in this sequence from its successor, however, yields a new sequence:

```
1001 1001 1001
```

which can be described by a simple, nongeneralized procedure:

```
TO CONSTANT /N/
10 OUTPUT /N/
END
```

(This is clearly a better rule than the generalized FIVERS for describing constant sequences)

Assuming CONSTANT has been entered in /PROCEDURE-BANK/, OCCAM, in its present form, can easily find the next term in the *difference sequence*, i.e., the sequence made up of the difference of successive terms in the given sequence:

```
←OCCAM
SEQUENCE?...1001 1001 1001
IS THE NEXT TERM 1001?...YES
THE WINNING PROCEDURE WAS CONSTANT
←
```

We can now hazard a successor for the given sequence itself. Our method is based upon a truth and a supposition. The truth is: each term in a sequence is expressible as the sum of its predecessor and the corresponding term in its difference sequence. Now our supposition: the next term will *probably* be the sum of the last term in the original sequence (3004 in the example) and the winning next term of the difference sequence (1001), that is, 4005. Since this looks like an effective strategy (after all, it seemed to work here, the first time we used it), let's put it into OCCAM. We'll modify OCCAM so that, if its existing strategies are unsuccessful, it will build the difference sequence, try to solve it, and use the solutions it finds to compute possible next terms for the original problem sequence.

We'll first write a procedure DIFF-SEQUENCE whose input is a sequence and whose output is the corresponding difference sequence. It simply outputs a sentence of differences of successive terms in the input sequence.

```
TO DIFF-SEQUENCE /SEQUENCE/
10 TEST EMPTYP BUTFIRST OF        (Have we subtracted all pairs?)
   /SEQUENCE/                     (Yes, done)
20 IF TRUE OUTPUT /EMPTY/            (If not, put the difference of
30 OUTPUT SENTENCE OF DIFFERENCE OF the first two terms in a
   (SECOND OF /SEQUENCE/) AND         sentence and repeat the
   (FIRST OF /SEQUENCE/)              process on the reduced
   AND DIFF-SEQUENCE OF BUTFIRST      sequence)
   OF /SEQUENCE/
END
```

DIFF-SEQUENCE will, of course, transform arithmetic sequences into constant sequences.

```
←PRINT DIFF-SEQUENCE OF "1 1002 2003 3004"
1001 1001 1001
←PRINT DIFF-SEQUENCE OF "66 77 88 99"
11 11 11
←PRINT DIFF-SEQUENCE OF "60 52 44 36 28"
-8 -8 -8 -8
←
```

The usefulness of DIFF-SEQUENCE does not stop here, though. Various other kinds of sequences also yield difference sequences that are easier to solve than the originals:

```
←PRINT DIFF-SEQUENCE OF "121 144 169 196"
23 25 27
←PRINT DIFF-SEQUENCE OF "1 12 123 1234"
11 111 1111
```

```
←PRINT DIFF-SEQUENCE OF "5 5 6 8 8 9"
0 1 2 0 1
←
```

We shall now use DIFF-SEQUENCE to make a difference sequence out of any sequence that the procedures in SCAN-P-LIST and SCAN-G-LIST fail to extrapolate. If this difference sequence is tractable, we will add its potential successor to the last term of the original sequence and submit the result to APPROVAL. If the extrapolation fails, we will try again with other potential successors of the difference sequence. Let's write a procedure SCAN-DIFF to do this. SCAN-DIFF differs from SCAN-P-LIST only in the additional input, the computation of /NEXT-TERM/, and the form of its winning procedure declaration.

```
TO SCAN-DIFF /SEQUENCE/ AND /DIFF-SEQUENCE/ AND /PROCEDURES/
10 TEST EMPTYP /PROCEDURES/
20 IF TRUE OUTPUT /EMPTY/
30 TEST TESTER OF /DIFF-SEQUENCE/ (Does the first procedure de-
    AND FIRST OF /PROCEDURES/      scribe the difference sequence?)
40 IF FALSE OUTPUT SCAN-DIFF OF   (If not, try the rest of the
    /SEQUENCE/ AND /DIFF-SEQUENCE/ bank)
    AND BUTFIRST OF /PROCEDURES/
50 MAKE "NEXT-TERM" SUM OF        (If so, the next term is the
    (LAST OF /SEQUENCE/) AND       last term in /SEQUENCE/ plus
    (EXECUTE OF FIRST OF           the next term of
     /PROCEDURES/ AND              /DIFF-SEQUENCE/)
    LAST OF /DIFF-SEQUENCE/)
60 TEST APPROVAL OF /NEXT-TERM/
70 IF TRUE PRINT SENTENCE OF
    "THE WINNING PROCEDURE WAS"
    AND SENTENCE OF FIRST OF
    /PROCEDURES/ AND "APPLIED TO  (Note the additional comment)
    THE DIFFERENCES OF THE TERMS"
80 IF TRUE OUTPUT FIRST OF
    /PROCEDURES/
90 OUTPUT SCAN-DIFF OF /SEQUENCE/
    AND /DIFF-SEQUENCE/ AND
    BUTFIRST OF /PROCEDURES/
END
```

Let's also write a related procedure SCAN-GEN-DIFF, for performing the same process, using generalized procedures.

```
TO SCAN-GEN-DIFF /SEQUENCE/ AND /DIFF-SEQUENCE/ AND /PROCEDURES/
10 TEST EMPTYP /PROCEDURES/
20 IF TRUE OUTPUT /EMPTY/
30 MAKE "WINNING-DUMMY" GENERAL-
    TESTER OF /DIFF-SEQUENCE/     (Test the first procedure against
    AND FIRST OF /PROCEDURES/      the difference sequence)
40 TEST EMPTYP /WINNING-DUMMY/
50 IF TRUE OUTPUT SCAN-GEN-DIFF  (If it fails, go on to the rest
    OF /SEQUENCE/ AND              of the list)
    /DIFF-SEQUENCE/ AND
    BUTFIRST OF /PROCEDURES/
60 MAKE "NEXT-TERM" SUM OF       (If it wins, make the next term
    (LAST OF /SEQUENCE/) AND       the sum of the last term in
    (EXECUTE-2 OF (FIRST OF        /SEQUENCE/ and the next term
    /PROCEDURES/) AND              of /DIFF-SEQUENCE/)
    (LAST OF /DIFF-SEQUENCE/)
    AND (/WINNING-DUMMY/)
70 TEST APPROVAL OF /NEXT-TERM/
80 IF TRUE PRINT SENTENCE OF
    "THE WINNING PROCEDURE WAS"
    AND SENTENCE OF FIRST OF
    /PROCEDURES/ AND SENTENCE OF
    /DUMMY/ AND "APPLIED TO THE
    DIFFERENCES OF THE TERMS"
90 IF TRUE OUTPUT FIRST OF /PROCEDURES/
100 OUTPUT SCAN-GEN-DIFF OF /SEQUENCE/
    AND /DIFF-SEQUENCE/ AND BUTFIRST
    OF /PROCEDURES/
END
```

To incorporate SCAN-DIFF and SCAN-GEN-DIFF in the extrapolation program, we need only add them as alternative tests in OCCAM. Lines 30 through 40 of OCCAM will now look like this:

```
30 TEST EMPTYP OF SCAN-P-LIST OF /SEQUENCE/ AND /PROCEDURE-BANK/
35 IF TRUE TEST EMPTYP OF SCAN-G-LIST OF /SEQUENCE/AND /GENERAL-
    BANK/
37 IF TRUE TEST EMPTYP OF SCAN-DIFF OF /SEQUENCE/ AND (DIFF-
    SEQUENCE OF /SEQUENCE/) AND /PROCEDURE-BANK/
39 IF TRUE TEST EMPTYP OF SCAN-GEN-DIFF OF /SEQUENCE/ AND
    (DIFF-SEQUENCE OF /SEQUENCE/) AND /GENERAL/BANK/
40 IF FALSE STOP
```

OCCAM is once again ready for testing. Let's start with our old problem:

```
←OCCAM
SEQUENCE?...1 1002 2003 3004
IS THE NEXT TERM 4005?...YES
THE WINNING PROCEDURE WAS CONSTANT APPLIED TO THE
DIFFERENCES OF THE TERMS
←
```

Let's show how OCCAM uses the differencing strategy to transform a variety of different kinds of sequences into tractable form:

```
←OCCAM
SEQUENCE?...75 60 45 30
IS THE NEXT TERM 15?...YES
THE WINNING PROCEDURE WAS CONSTANT APPLIED TO THE
DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...2 4 8
IS THE NEXT TERM 16?...NO
IS THE NEXT TERM 24?...YES
THE WINNING PROCEDURE WAS SQUARE APPLIED TO THE
DIFFERENCES OF THE TERMS
```

Frequently OCCAM will apply the strategy of differencing usefully to generalized procedures:

```
←OCCAM
SEQUENCE?...2 5 10 17 26
IS THE NEXT TERM 37?...YES
THE WINNING PROCEDURE WAS ADD-TWO FOR /DUMMY/ EQUALS 2
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...1 12 123 1234
IS THE NEXT TERM 12345?...YES
THE WINNING PROCEDURE WAS GROW-ONE FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS
←
```

OCCAM can now make a guess at the successor of all sequences whose difference sequences can be described by any of the procedures in the banks. Thus, it can solve such sequences as these:

```
←OCCAM
SEQUENCE?...1 1 2 2 3 3 4
IS THE NEXT TERM 4?...YES
THE WINNING PROCEDURE WAS MOD-FOUR FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...50 66 74 78
IS THE NEXT TERM 80?...YES
THE WINNING PROCEDURE WAS DIV-TWO FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...0 1 3 9 33
IS THE NEXT TERM 153?...YES
THE WINNING PROCEDURE WAS NEXT-FACTORIAL FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...1 2 6 15 31
IS THE NEXT TERM 56?...YES
THE WINNING PROCEDURE WAS SQUARES FOR /DUMMY/ EQUALS 2
APPLIED TO THE DIFFERENCES OF THE TERMS
```

OCCAM can now solve difficult and esoteric sequences as well as familiar, "natural" types; a high-school student is not likely to ask for a successor to the sequence 18 20 44 724..., for example, though OCCAM can give one. Also, the existing (generalizing and difference-transforming) strategies using the relatively small set of procedures presently in the banks, will often yield a large number of plausible extrapolations. Thus, for example, OCCAM can now extrapolate over a thousand possible successors for each sequence made up of two random numbers (though it is by no means assured of getting the "right" one).

The strategy of transforming sequences by taking differences of successive terms is one of a set of such effective strategies. Others of this kind include transforming a given sequence into the sequence comprised of its alternate terms, its partial sums (products, quotients) over one or more preceding terms, etc. Problems to incorporate these strategies are given in the associated problem sequel.

As the extrapolation program becomes larger both in the number of procedures it has explicitly to consider and the number of different strategies it has for modifying procedures or sequences, it can become unwieldy or inefficient. Under these circumstances, the use of higher order strategies or of administrative procedures, to improve the operation of OCCAM, becomes important. Some ideas for designing these kinds of facilities are also considered in the problem set.

Thus far, OCCAM has been developed to work in the single domain of sequences of *numbers*. With straightforward modification, it can be adapted for use with *letter sequences* or as the basis of a simple *function-guessing* situation. Further, using the same methods already built into OCCAM, the "inverse" program can be built for *generating* various kinds of sequences for the *user* to extrapolate. These extensions also are considered in the problem set.

Rather than studying further problem-solving strategies in the context of sequence guessing, we shall next study new kinds of strategies in yet another, distinctly different problem-solving situation, traversing a maze.

3. MAZES

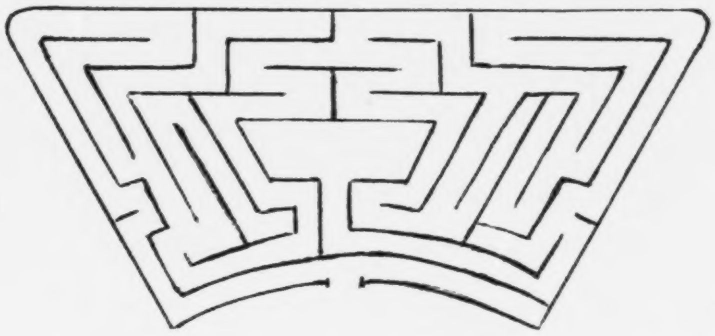
3.1 Introduction - A Minimal History of Mazes

The most famous maze of antiquity, the Labyrinth, was a gigantic walled structure reputedly on the island of Crete designed by a man named Daedalus. It had two openings to the outside world between which lay not only an incredibly complex series of corridors, loops, blind alleys, and traps, but the "man-eating" Minotaur as well. Only three men are said to have escaped from the Labyrinth. Resourceful Daedalus (who, it seems, couldn't otherwise master his own creation) and his son Icarus, flew out the top with wings fashioned from wax and bird feathers; the clever hero Theseus trailed a thread behind him and, after he killed the Minotaur, he followed the thread back to the entrance and freedom.

After the Cretan empire fell and the Labyrinth was destroyed, real mazes fell somewhat from favor. They flourished once again, however, in 17th, 18th, and 19th century England, in palace gardens, constructed as carefully clipped barriers of impenetrable shrubbery. The plan of the maze at Hampton Court Palace, perhaps the most intricate of such mazes, is shown in Figure 1.

Most modern mazes are both more modest and less treacherous. Earlier in this century behavioral scientists found simple mazes useful for testing certain learning patterns with various animals. Unlike the English or Cretan varieties, these mazes generally had no exits -- the mouse or rat was simply dropped into the maze at some starting place and lifted out when it found the food.

Our task in the following pages will be to put ourselves in the place of the Minotaur's prey, or a bewildered Englishman, or a hungry white mouse -- only we will program our way through by writing LOGO procedures, first simply to direct our motion around mazes, and then to develop and improve *strategies* for solving maze problems.



Maze in the
gardens at Hampton
Court Palace (1880)

Figure 1.

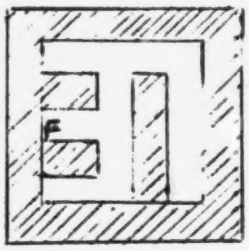
3.2 Setting up Mazes

We first need to specify the form in which mazes are stored and referred to by the computer. A good start is to divide the surface of a maze into one-character squares, some of which are open to travel and some of which are impenetrable areas, that is "walls". Let's specifically denote a wall square by writing a "+", and an open square by a "0". To start with, let's consider mazes which are completely closed to the outside and have only one "finish" square, which we denote "F". We can start at any open square and our goal is to get to square "F". Further, denoting an outside wall by an empty square, we do not need to bother typing the last wall of +'s around each maze. With these conventions, one of these mazes might look like this:

Maze-1:

```
00000
++0+0
F00+0
++0+0
000+0
```

corresponding to a walled structure pictured like this:



Now that we have specified our conventions, let's write a procedure that will accept a maze (storing it by name for future reference), and another procedure that will take the name of a maze and draw the maze associated with it. We will need to represent our mazes as LOGO objects -- words or sentences -- not directly as two-dimensional objects. One way of storing mazes is to make each row one word of a sentence. Thus, the maze pictured above can be stored as the sentence containing all the lines in the order they are drawn:

/MAZE-1/ is "00000 ++0+0 F00+0 ++0+0 000+0"

In order to accept a maze, then, we will need a procedure that requests one row at a time and builds a sentence out of these word-rows. The procedure ACCEPT-LINES does just this, terminating when, instead of a row of +'s and 0's (possibly with an F), the word "END" is typed in.

```
TO ACCEPT-LINES
10 MAKE "LINE" REQUEST            (Get the next line)
20 TEST IS /LINE/ "END"           (All done?)
30 IF TRUE OUTPUT /EMPTY/         (If so, terminate)
40 OUTPUT SENTENCE OF /LINE/      (Otherwise, append new line to
   AND ACCEPT-LINES               the previous lines)
END
```

ACCEPT-LINES will build an appropriate sentence but does not store it. This is easily accomplished with the procedure ACCEPT-MAZE, whose input is the name to be given to the maze.

```
TO ACCEPT-MAZE /NAME/
10 MAKE /NAME/ ACCEPT-LINES      (Assign the input name to the
END                               typed in maze)
```

Using ACCEPT-MAZE, we can now type mazes into the computer and retrieve them subsequently.

```
←ACCEPT-MAZE "MAZE-1"
:00000
:++0+0
:F00+0
:++0+0
:000+0
:END
←PRINT /MAZE-1/
00000 ++0+0 F00+0 ++0+0 000+0
```

When we look at a maze, however, we want to see the two-dimensional structure, not the LOGO sentence which represents it. To this end we will write procedures SHOW-MAZE and SHOW-LINES that do the inverse of which was done by ACCEPT-MAZE and ACCEPT-LINES, i.e., they make a two-dimensional drawing of a specified maze.

```
TO SHOW-LINES /MAZE/
10 TEST EMPTYP OF /MAZE/             (Is the maze-sentence exhausted?)
20 IF TRUE STOP                      (If so, stop)
30 PRINT FIRST OF /MAZE/             (Otherwise, print the first row)
40 SHOW-LINES OF BUTFIRST OF /MAZE/  (And get the rest)
END
```

```
TO SHOW-MAZE /NAME/
10 SHOW-LINES OF (THING OF /NAME/)  (Diagram the maze-sentence
END                                  specified by /NAME/)
```

Thus:

```
←SHOW-MAZE "MAZE-1"
00000
++0+0
F00+0
++0+0
000+0
←
```

We can now store and retrieve a maze as a whole, but in order to go further we need a means of specifying position within the maze. We must refer to individual squares, both to identify our position within the maze and also, when moving, to find permissible adjacent squares for our next move. A standard method for labeling grids is to number rows and columns as shown with the MAZE-1 example:

```
  12345
1 00000
2 ++0+0
3 F00+0
4 ++0+0
5 000+0
```

A particular position within the grid is then referred to as a pair of numbers -- the column-number and row-number, respectively. Thus, in the above example, the finish is at "1 3"; first column, third row.

In moving through a maze we must be able to determine what kinds of squares are adjacent to the one being occupied. Certainly our next move will be very different if a given adjacent square contains an "F" rather than a "+". A procedure GET-STATUS, which does this, needs two inputs, the position of interest and the maze name, and outputs the character contained in the square thus specified. We want, for example, the following result.

```
←PRINT GET-STATUS OF "1 3" AND "MAZE-1"
F
←
```

In the example above, GET-STATUS would first isolate the third column which is the third word of the maze-sentence "MAZE-1". A good way to attack this problem is to write a subprocedure, NTH OF /N/ AND /EXPRESSION/ that outputs the /N/th word of the sentence, /EXPRESSION/. This can be written as follows:

```
TO NTH /N/ AND /EXPRESSION/
10 TEST IS /N/ 1                    (Do we want the first word?)
20 IF TRUE OUTPUT (FIRST OF         (If so, output it)
     /EXPRESSION/)
30 OUTPUT NTH OF                    (If not, diminish /N/ remove
     (DIFF OF /N/ AND 1)            one word from /EXPRESSION/,
     (BUTFIRST OF /EXPRESSION/)     and try again)
END
```

Once the right word has been isolated, GET-STATUS will then have to extract the proper character from that word; in our example, the first character. The procedure to do this is identical to NTH, since in LOGO FIRST and BUTFIRST work on words and sentences in precisely parallel ways. So our subprocedures are all written and GET-STATUS follows easily:

```
TO GET-STATUS /POSITION/ AND /MAZE-NAME/
10 OUTPUT NTH OF (FIRST OF /POSITION/)      (The proper column)
     AND (NTH OF (LAST OF /POSITION/)       (The proper row)
     AND (THING OF /MAZE-NAME/))            (The maze-sentence itself)
END
```

Let's try out GET-STATUS.

```
←PRINT GET-STATUS OF "1 1" AND "MAZE-1"
0
←PRINT GET-STATUS OF "1 0" AND "MAZE-1"
(no response until break key terminates the computation)
```

If we try to use GET-STATUS, as it now stands, for positions past the left-hand or top edge of a maze, we will run into trouble.

The problem is that NTH OF 0 AND /ANYTHING/ goes on indefinitely. Let's take care of this situation by adopting the convention that the type (or status) of outside squares will be /EMPTY/. Then we can fix NTH by adding the following lines to it:

```
4 TEST GREATERP 1 AND /N/
6 IF TRUE OUTPUT /EMPTY/
```

Now that we can build mazes and examine the status of any position in them, we are ready to work on the problem of moving through a maze. Imagine a traveler moving through it square-by-square. The current position of this occupant, say a mouse, can be represented by a pair of coordinates (i.e., a sentence of column number and row number). We need a name for this position, the current square, and "MOUSE" is a fine choice. To put the mouse in the top right-hand corner of the maze, we need only type:

```
←MAKE
    NAME:  "MOUSE"
    THING:  "5 1"
←
```

Given the mouse's position, /MOUSE/, we can easily determine the coordinates of the squares to which it might move. The square directly in front of it (meaning up), for example, is specified by the sentence of FIRST OF /MOUSE/ and DIFFERENCE OF (LAST OF /MOUSE/) AND 1. By similarly adding or subtracting 1 to the first or last word of /MOUSE/, we can immediately move the mouse to the left, right, front, or back. Let's actually write procedures to do this, calling them, respectively, LEF, RIGH, FRON, BAC (we use these names because the words LEFT, RIGHT, FRONT, and BACK already are used as special built-in LOGO commands for moving it's robot "Turtle").

```
TO LEF /MOUSE/
10 OUTPUT SENTENCE OF (DIFFERENCE OF FIRST OF /MOUSE/
   AND 1) AND (LAST OF /MOUSE/)
END
```

```
TO RIGH /MOUSE/
10 OUTPUT SENTENCE OF (SUM OF FIRST OF /MOUSE/ AND 1)
   AND (LAST OF /MOUSE/)
END
```

```
TO FRON /MOUSE/
10 OUTPUT SENTENCE OF (FIRST OF /MOUSE/) AND
  (DIFFERENCE OF LAST OF /MOUSE/ AND 1)
END
```

```
TO BAC /MOUSE/
10 OUTPUT SENTENCE OF (FIRST OF /MOUSE/) AND
   (SUM OF LAST OF /MOUSE/ AND 1)
END
```

If the mouse is still in the top right-hand corner of /MAZE-1/, then we can move it one square to the left by typing:

```
←MAKE "MOUSE" LEF OF /MOUSE/
←PRINT /MOUSE/
4 1
←
```

3.3 Moving a Mouse Through a Maze

With our four basic moving procedures, we can now build a procedure GO-THROUGH that uses them to move the mouse through a maze, one square at a time. We'll start with a very simple GO-THROUGH.

Let's start by assuming that a strategy consists of a single procedure (FRON, BAC, LEF, or RIGH) that we apply to the mouse's position until either (1) it runs into a wall, or (2) it finds the food. GO-THROUGH, then, moves the mouse according to the rule specified by the procedure and checks the status of the new position to see if either of the terminating conditions are satisfied. If not, GO-THROUGH announces the new position and repeats. Thus, GO-THROUGH needs three inputs: /MOVE/ which specifies which procedure (such as FRON) is to be followed, /MOUSE/ which specifies the mouse's current position, and, of course, /MAZE-NAME/ to identify the maze.

```
TO GO-THROUGH /MOVE/ AND /MOUSE/ AND /MAZE-NAME/
10 MAKE "MOUSE" EXECUTE OF /MOVE/ AND /MOUSE/
20 MAKE "STATUS" GET-STATUS OF /MOUSE/ AND /MAZE-NAME/
30 TEST IS /STATUS/ "F"
40 IF TRUE PRINT SENTENCE OF "I AM AT"              (Is the food found?)
   AND SENTENCE OF /MOUSE/ "AND I FOUND
   THE CHEESE!"
50 IF TRUE STOP
60 TEST EITHER (IS /STATUS/ "+")
   (EMPTYP /STATUS/)
70 IF TRUE STUCK                                    (Is this a wall?)
80 PRINT SENTENCE OF "AT" AND /MOUSE/               (Announce new square)
90 GO-THROUGH /MOVE/ AND /MOUSE/ AND /MAZE-NAME/        (Repeat)
END
```

GO-THROUGH uses two subprocedures, EXECUTE and STUCK. STUCK gives a repeated, end-of-the-line message that the mouse has run into a wall:

```
TO STUCK
10 PRINT "STUCK!"
20 STUCK
END
```

EXECUTE is the procedure that actually applies the specified move to the mouse's position. It incorporates the command DO in order to use /MOVE/ as a procedure applied to /MOUSE/.

```
TO EXECUTE /PROCEDURE/ AND /INPUT/
10 DO SENTENCE OF "OUTPUT" AND SENTENCE OF
   /PROCEDURE/ AND /INPUT/
END
```

Let's try GO-THROUGH with /MAZE-1/ for illustration. First, here is /MAZE-1/.

```
←SHOW-MAZE "MAZE-1"
00000
++0+0
F00+0
++0+0
000+0
←GO-THROUGH "LEF" AND "3 3" AND "MAZE-1"
AT 2 3
I AM AT 1 3 AND I FOUND THE CHEESE!
←GO-THROUGH "FRON" AND "3 5" AND "MAZE-1"
AT 3 4
AT 3 3
AT 3 2
AT 3 1
STUCK!
STUCK!
STUCK!
(BREAK)                         (The user stops STUCK)
I WAS AT LINE 10 IN STUCK
←
```

So far we have defined a very limited mouse; it only travels in the one, given, direction and thus can only find the cheese if placed directly in line with it with no intervening walls. What we really want is a GO-THROUGH that will handle a *list* of possible moves. That way, if the first type of move fails (i.e., threatens to run the mouse into a wall), the next one can be executed, or the one after that, and so on until the strategy list has been exhausted. This will clearly give a great deal more flexibility.

In order to process a strategy list without getting stuck too soon, we cannot use the EXECUTE in GO-THROUGH directly. Instead, want to evaluate each possible next position before it is given to GO-THROUGH to insure that it is not a wall square. This intermediary role will be played by a new procedure, NEXT, which outputs the mouse's next permissible position, if there is one. GO-THROUGH thus becomes:

```
TO GO-THROUGH /STRATEGIES/ AND /MOUSE/ AND /MAZE-NAME/
10 MAKE "MOUSE" NEXT OF /STRATEGIES/ AND /MOUSE/ AND /MAZE-NAME/
20 MAKE "STATUS" GET-STATUS OF /MOUSE/ AND /MAZE-NAME/
30 TEST IS /STATUS/ "F"
40 IF TRUE PRINT SENTENCE OF "I AM AT" AND SENTENCE
   OF /MOUSE/ AND "AND I FOUND THE CHEESE!"
50 IF TRUE STOP
60 PRINT SENTENCE OF "AT" AND /MOUSE/
70 GO-THROUGH /STRATEGIES/ AND /MOUSE/ AND /MAZE-NAME/
END
```

The subprocedure NEXT tries the first kind of move on its strategy list. If this is a permissible move, NEXT outputs the new position of the mouse. If the first move fails, then NEXT tries successive moves on the strategy list, either until one of them gives a legitimate move, or until the list is exhausted, in which case the mouse is STUCK.

```
TO NEXT /STRATEGIES/ AND /MOUSE/ AND /MAZE-NAME/
10 TEST EMPTYP /STRATEGIES/       (Have we exhausted the strategies?)
20 IF TRUE STUCK                  (If so, we're stuck)
30 MAKE
    "TRIAL"
    EXECUTE OF FIRST OF /STRATEGIES/
    AND /MOUSE/
40 MAKE
    "STATUS"
    GET-STATUS OF /TRIAL/ AND /MAZE-NAME/
50 TEST EITHER
    IS /STATUS/ "+"
    EMPTYP /STATUS/               (Is it a wall?)
60 IF FALSE OUTPUT /TRIAL/        (If not, make the move)
70 OUTPUT NEXT OF
    (BUTFIRST OF /STRATEGIES/)
    AND /MOUSE/
    AND /MAZE-NAME/               (Otherwise, try a new strategy)
END
```

Now we can try some more runs through the maze using more complex strategies. For example, we can try to keep going to the right; when that is not possible, move upwards, to the left, and, if all else fails, downwards:

```
←SHOW-MAZE "MAZE-1"
00000
++0+0
F00+0
++0+0
000+0
←GO-THROUGH "RIGH FRON LEF BAC" AND "1 5" AND "MAZE-1"
AT 2 5                 (going RIGH)
AT 3 5                     "
AT 3 4                 (forced to go FRON)
AT 3 3                      "
AT 3 2                      "
AT 3 1                      "
AT 4 1                 (free to go RIGH)
AT 5 1                      "
AT 4 1                 (forced to go LEF)
AT 5 1                 (free to go RIGH)
AT 4 1                 (etc. - we are oscillating)
BREAK
I WAS AT LINE 50 IN NEXT
←
```

We now have considerably more flexibility than with our previous version of GO-THROUGH -- our mouse can now trace complicated routes through a maze. A difficulty remains -- whatever the order of the moves on the strategy list, the mouse will eventually get stuck in an oscillation, of the same sort as in our preceding example. If the finish comes before the squares on which the oscillation sets in, all is well, but, clearly, this is not true at least in some cases. The reason for the oscillation is that for any maze, the mouse has eventually to change its direction. As soon as it has done so, the original direction of motion is again possible, leading to an infinite repetition of these two moves. If the first two moves on the strategy list are opposite, as in FRON BAC RIGH LEF for example, the mouse goes forward until it reaches an obstacle. It then goes backward in which case forward is again possible, backwards, forwards, backwards, forwards, etc. The argument is the same if the first move cannot be FRON. If it can be BAC, the oscillation sets in immediately.

If neither FRON or BAC is possible, the same line of reasoning gets one in trouble still, though in this case the oscillation may involve RIGH and LEF. The only mazes which completely avoid this difficulty must be completely traversable only by two commands -- one of the pair FRON,BAC and one of the pair RIGH, LEF.

3.4 Relative and Absolute Motion

We saw in the last section that no combination of moves can form a generally successful strategy. The reason for this lies in the *absolute* nature of our definitions of directions. That is, directions are defined with respect to the *maze* -- FRON means upward along a column, RIGH along a row in the direction of increasing row numbers, etc. -- not the *mouse*. Directions relative to the direction the mouse is facing are much more useful in maze tracing. Assume then, that we assign the mouse a face, i.e., a front, and therefore, a back, right, and left. We should like to say - go to *your* left, turn around, etc.

If FRON and BAC referred to these *relative* directions, then a strategy list FRON BAC would cause the mouse to go forward along a corridor until it met a wall, turn around, then return. This is clearly a reasonable way to deal with dead ends. In the same situation the absolute definition of direction gives rise to a perpetual oscillation at the end of the corridor. Thus, we want to redefine our definitions of directions so that they are relative to the way the mouse is facing.

As now defined /MOUSE/ just represents the mouse's current square - there is no information about its orientation included. One way to give orientation is to use, instead of a one-square mouse, a two-character mouse whose position and orientation are represented by a pair of adjacent points - a /NOSE/ and a /TAIL/. Let's start rewriting our maze procedures to enable them to handle such a mouse.

Redoing GO-THROUGH is fairly simple. We can still regard the current position as one point, the position of /NOSE/, since /TAIL/ is just the last position reported. Computing the new /NOSE/ will still be handled by NEXT, and the new /TAIL/ is just the old /NOSE/:

```
TO GO-THROUGH /STRATEGIES/ AND /NOSE/ AND /TAIL/ AND /MAZE-NAME/
10 MAKE "OLD-NOSE" /NOSE/
20 MAKE "NOSE" NEXT OF /STRATEGIES/ AND /NOSE/ AND /TAIL/
    AND /MAZE-NAME/
30 MAKE "TAIL" /OLD-NOSE/
40 MAKE "STATUS" GET-STATUS OF /NOSE/ AND /MAZE-NAME/
50 TEST IS /STATUS/ "F"
60 IF TRUE PRINT SENTENCE OF "I AM AT" AND SENTENCE OF /MOUSE/
    AND "AND I FOUND THE CHEESE!"
70 IF TRUE STOP
80 PRINT SENTENCE OF "AT" AND /NOSE/
90 GO-THROUGH /STRATEGIES/ AND /NOSE/ AND /TAIL/ AND /MAZE-NAME/
END
```

Fixing NEXT is even easier. Beyond the title and the recursion line (Line 70), the only change is that we must now use a three-input EXECUTE for computing /TRIAL/:

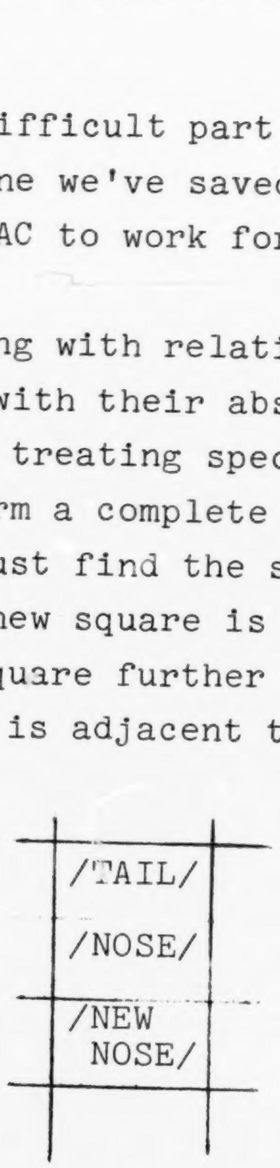
```
TO NEXT /STRATEGIES/ AND /NOSE/ AND /TAIL/ AND /MAZE-NAME/
10 TEST EMPTYP /STRATEGIES/
20 IF TRUE STUCK
30 MAKE "TRIAL" EXECUTE-2 OF (FIRST OF /STRATEGIES/)
    AND /NOSE/ AND /TAIL/
40 MAKE "STATUS" GET-STATUS OF /TRIAL/ AND /MAZE-NAME/
50 TEST EITHER (IS /STATUS/ "+") (EMPTYP /STATUS/)
60 IF FALSE OUTPUT /TRIAL/
70 OUTPUT NEXT OF (BUTFIRST OF /STRATEGIES/) AND
    /NOSE/ AND /TAIL/ AND /MAZE-NAME/
END
```

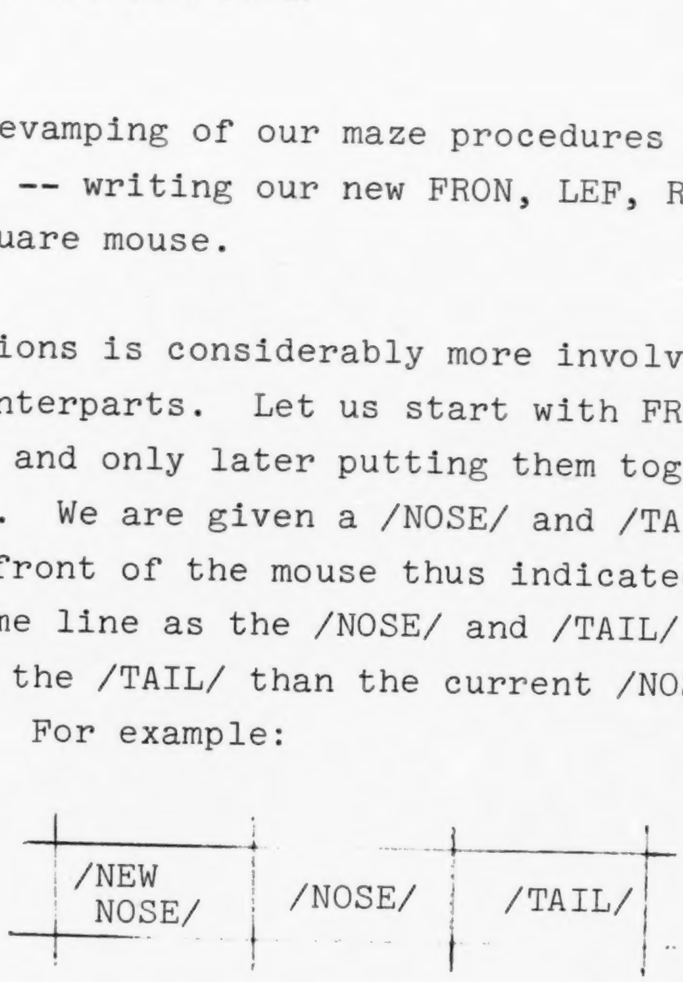
The subprocedure EXECUTE-2 is a simple extension of EXECUTE:

```
TO EXECUTE-2 /PROCEDURE/ AND /INPUT-1/ AND /INPUT-2/
10 DO SENTENCE OF "OUTPUT" AND SENTENCE OF /PROCEDURE/
    AND SENTENCE OF /INPUT-1/ AND /INPUT-2/
END
```

The difficult part of this revamping of our maze procedures is the one we've saved for last -- writing our new FRON, LEF, RIGH, and BAC to work for a two-square mouse.

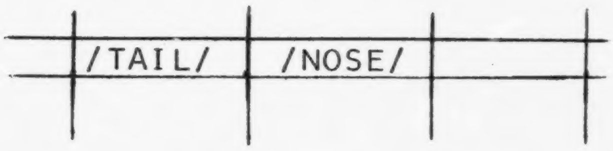
Dealing with relative directions is considerably more involved than with their absolute counterparts. Let us start with FRON, first treating special cases and only later putting them together to form a complete algorithm. We are given a /NOSE/ and /TAIL/ and must find the square in front of the mouse thus indicated. This new square is on the same line as the /NOSE/ and /TAIL/ but one square further away from the /TAIL/ than the current /NOSE/, which is adjacent to /TAIL/. For example:

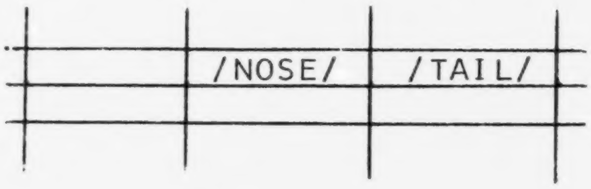




In order to give the computer an algorithm for finding the square in front of a given two-square mouse, we must turn this graphic view of front into a numerical one. In other words, we must give the coordinates of the "front move" in terms of the coordinates /NOSE/ and /TAIL/. Let us first consider a horizontal mouse and later a vertical one.

There are two kinds of horizontal mice, as shown below

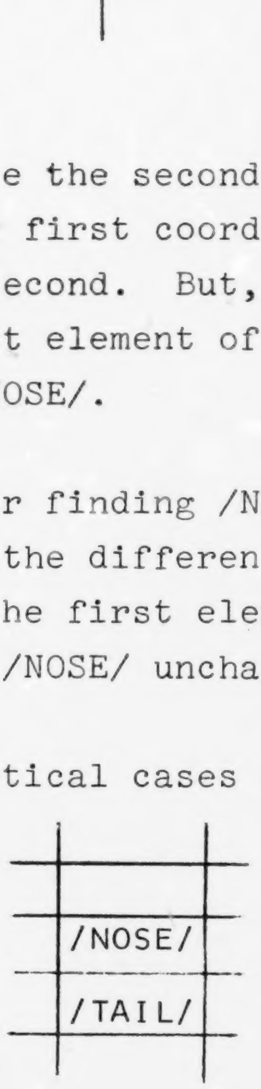


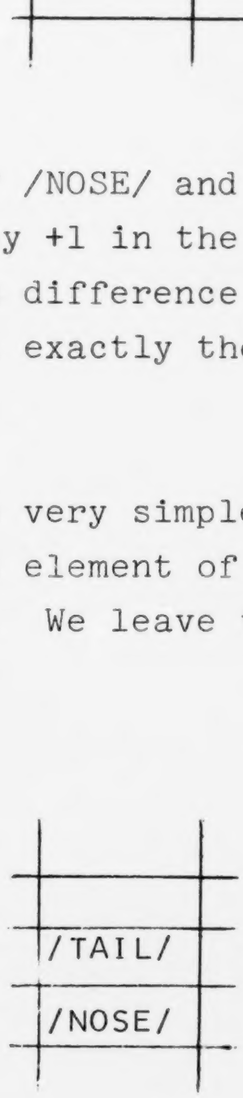


In each case the second coordinates of /NOSE/ and /TAIL/ are the same, their first coordinates differ by +1 in the first case and -1 in the second. But, if we add this difference in each case to the first element of /NOSE/, we get exactly the square in front of /NOSE/.

The rule for finding /NEWNOSE/ is thus very simple. RULE 1 We simply add the difference of the first element of /NOSE/ and /TAIL/ to the first element of /NOSE/. We leave the second element of /NOSE/ unchanged.

The two vertical cases are





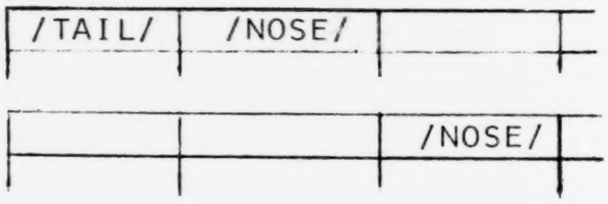
Here, the first elements are the same, the second coordinates differ by 1 and by -1 which again are the amounts the second coordinate is to be changed in each case. Thus, for vertical mice: RULE 2 we leave the first coordinate of /NOSE/ unchanged. We add the difference of SECOND /NOSE/ and SECOND /TAIL/ to the second coordinate of /NOSE/.

We now have a rule for finding the square in front of a horizontal and of a vertical mouse. To combine these two, to work for any mouse, we note that RULE 1 has absolutely no effect on a vertical mouse. This is true because the change given by RULE 1 involves only a difference of first coordinates. But a vertical mouse has the first coordinates of /NOSE/ and /TAIL/ equal. RULE 2 has no effect on horizontal mice for a similar reason. Thus we can combine the two rules into one, and only the relevant part will be non-zero for each of the two cases we have considered.

```
TO FRON /NOSE/ /TAIL/
10 OUTPUT SENTENCE
    SUM FIRST /NOSE/ DIFFERENCE (FIRST /NOSE/ FIRST /TAIL/)
    SUM SECOND /NOSE/ DIFFERENCE (SECOND /NOSE/ SECOND /TAIL/)
END
```

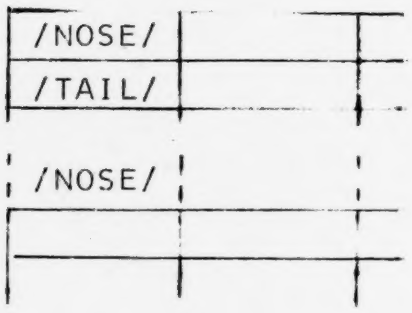
Then,

```
←PRINT FRON "2 1" "1 1"

3 1
←
```



```
←PRINT FRON "1 1" "1 2"

1 0
←
```

(Result: i.e., the new nose position is outside our normal grid)



We take the easiest case, BAC, next. To go backwards means simply that /NOSE/ replaces /TAIL/ (and /TAIL/ is the previous /NOSE/). Thus,

```
TO BAC /NOSE/ /TAIL/
10 OUTPUT /TAIL/
END
```

The formulas for calculating LEF and RIGH can be determined as with FRON by treating the four distinct cases that arise, and this results in:

LEF:
NEW-X = $X_N+(Y_N-Y_T)$
NEW-Y = $Y_N+(X_N-X_T)$

RIGH:
NEW-X = $X_N+(Y_T-Y_N)$
NEW-Y = $Y_N+(X_T-X_N)$

where X means first coordinate and Y stands for second coordinate. The subscripts N and T stand for /NOSE/ and /TAIL/ and NEW-X and NEW-Y denote the coordinates of the new nose position.

These rules are very similar to the ones we derived for FRON. Going right or left, however, means a change in direction, thus we see that the difference of second coordinates gives the change in first coordinates and the change in first coordinates helps give the new second coordinate.

The procedures follow easily from the formulas:

```
TO LEF /NOSE/ AND /TAIL/
10 OUTPUT SENTENCE OF (SUM OF (FIRST OF /NOSE/) AND
    (DIFF OF LAST OF /NOSE/ AND LAST OF /TAIL/)) AND
    (SUM OF (LAST OF /NOSE/) AND (DIFF OF FIRST
      OF /NOSE/ AND FIRST OF /TAIL/))
END
```

```
TO RIGH /NOSE/ AND /TAIL/
10 OUTPUT SENTENCE OF (SUM OF (FIRST OF /NOSE/) AND
    (DIFF OF LAST OF /TAIL/ AND LAST OF /NOSE/)) AND
    (SUM OF (LAST OF /NOSE/) AND (DIFF OF FIRST
      OF /TAIL/ AND FIRST OF /NOSE/))
END
```

Now we should be able to design a strategy for bearing left throughout a maze. What we want to do at every square is turn left if possible, if not go straight, if not go right, if not about-face. In short, the strategy is to "keep to the left". Let's try it:

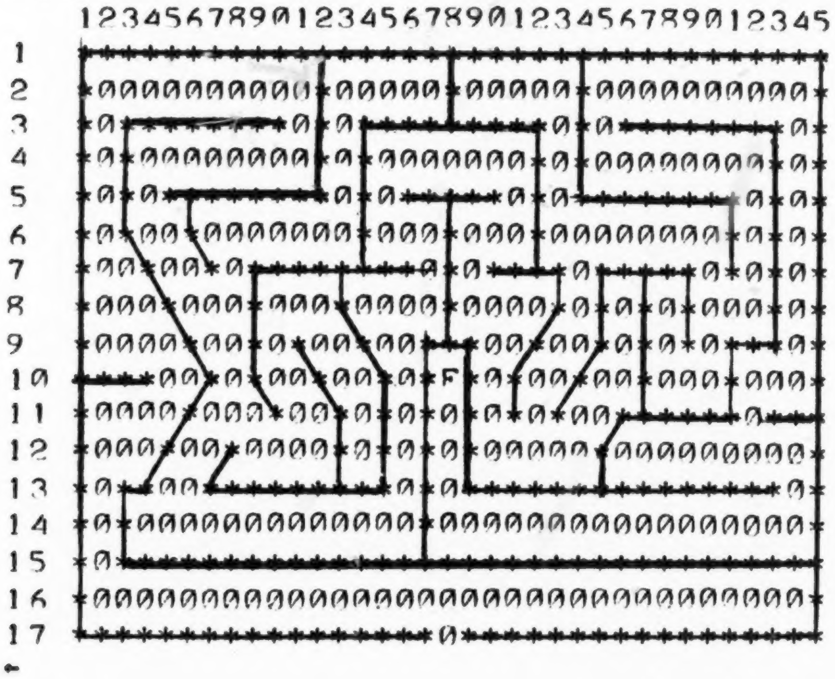
```
←SHOW-MAZE "MAZE-1"
00000
++0+0
F00+0
++0+0
000+0
←MAKE "NOSE" "5 4"
←MAKE "TAIL" "5 5"
←MAKE "STRATEGIES" "LEF FRON RIGH BAC"
←GO-THROUGH /STRATEGIES/ AND /NOSE/ AND /TAIL/ AND "MAZE-1"
AT 5 3        (forced to go FRON)
AT 5 2
AT 5 1
AT 4 1        (free to go LEF)
AT 3 1        (forced to go FRON)
AT 3 2        (free to go LEF)
AT 3 3        (forced to go FRON)
AT 3 4
AT 3 5
AT 2 5        (forced to go RIGH)
AT 1 5        (free to go FRON)
AT 2 5        (BAC)
AT 3 5        (FRON)
AT 3 4        (LEF)
AT 3 3        (FRON)
AT 2 3        (LEF)
I AM AT 1 3 AND I FOUND THE CHEESE!
←
```

Keeping to the left (right) means hugging the wall to the left (right) which ensures that the entire surface of the wall will be traversed. (If the wall has more than one side, only the side the mouse starts against will be thus traversed.) This strategy is a very generally useful one. It will work for any maze in which the finish square is against the same side of the same wall the mouse starts against. In particular, if all the walls are connected and the path is one square wide at all points, all accessible points in the maze would eventually be traversed by this strategy. (Squares which are "walled off" would be the only ones not reached.) The Hampton Court maze is of this "connected" type and would easily be traversed using this strategy. To accommodate it to the data type our programs deal with, we have to bend and stretch it a little, with the result shown below.

```
←SHOW-MAZE "HAMPTON"
    12345678901234567890123456789012345
1   +++++++++++++++++++++++++++++++++++
2   +0000000000+00000+00000+0000000000+
3   +0++++++++0+0+++++++++0+0++++++++0+
4   +0+00000000+0+0000000+0+00000000+0+
5   +0+0++++++++0+0+++++0+0++++++++0+0+
6   +0+00+0000000+000+000+00000000+0+0+
7   +00+00+0++++++++0+0++++0+++++0+0+0+
8   +000+000+000+0000+0000+0+0+0+000+0+
9   +0000+00+0+00+00+++00+00+0+0+0+++0+
10  ++++00+0+00+00+0+F+0+00+00+000+000+
11  +0000+000+00+0+0+0+0+0+00++++++0+++
12  +000+00+0000+0+0+0+00000+000000000+
13  +0++00+++++++++0+0+++++++++++++++0+
14  +0+0000000000000+00000000000000000+
15  +0+++++++++++++++++++++++++++++++++
16  +000000000000000000000000000000000+
17  +++++++++++++++++0+++++++++++++++++
←
```

The following outlined version is included to help in seeing the correspondence with the original drawing (Figure 1).

```
←SHOW-MAZE "HAMPTON-COURT"
```


12345678901234567890123456789012345
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
F

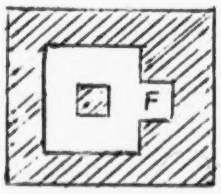
```
←
```

```
←MAKE "NOSE" "18 16"
←MAKE "TAIL" "18 17"
←MAKE "STRATEGIES" "LEF FRON RIGH BAC"
←GO-THROUGH /STRATEGIES/ /NOSE/ /TAIL/ "HAMPTON"
AT 17 16
AT 16 16
AT 15 16
AT 14 16
AT 13 16
AT 12 16
AT 11 16
AT 10 16
AT 9 16
AT 8 16
AT 7 16
AT 6 16
AT 5 16
AT 4 16
AT 3 16
AT 2 16
AT 2 15
AT 2 14
 : : :
 : : :
```

```
 :  :  :
 :  :  :
AT 24 14
AT 23 14
AT 22 14
AT 21 14
AT 20 14
AT 19 14
AT 18 14
AT 18 13
AT 18 12
AT 18 11
I AM AT 18 10 AND I FOUND THE CHEESE!
←
```

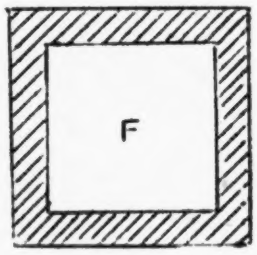
3.5 New Strategies

Using strategies like those just shown, the mouse can solve a large variety of mazes. It's worth spending some time experimenting with these strategies, however, because there are large classes of apparently simple mazes for which they fail. Consider the following simple example:



```
←ACCEPT-MAZE "MAZE-2"
:000+
:0+0F
:000+
:END
←MAKE "NOSE" "1 1"
←MAKE "TAIL" "1 2"
←MAKE "STRATEGIES" "RIGH FRON LEF BAC"
←GO-THROUGH /STRATEGIES/ AND /NOSE/ AND /TAIL/ AND "MAZE-2"
AT 2 1       (RIGH)
AT 3 1       (FRON)
AT 3 2       (RIGH)
AT 3 3       (FRON)
AT 2 3       (RIGH)
AT 1 3       (FRON)
AT 1 2       (RIGH)
AT 1 1       (FRON)
AT 2 1       (RIGH)
AT 3 1       (FRON)
(BREAK)
I WAS AT LINE 10 IN EXECUTE-2
←
```

The mouse, in this example, keeps hugging the wall to its right which is the isolated center square. Hugging a wall, however, only works, as we mentioned earlier, if the start and finish lie along the same wall. (Thus, our strategy fails, and in some sense this maze is actually more complicated than a connected maze, no matter how large.) This is not to say that all strategies based on FRON, LEF, RIGH, BAC will fail for this maze. In fact, bearing to the left, rather than the right, results in the mouse hugging the outside wall and thereby finding the cheese. But there are mazes, such as the following, for which no combination of our four basic moves will form a winning strategy unless the mouse starts off in line with the finish square or, in some cases, right next to it.



```
←ACCEPT-MAZE "MAZE-3"
:00000
:00000
:00F00
:00000
:00000
:END
←MAKE "NOSE" "2 3"
←MAKE "TAIL" "1 3"
←MAKE "STRATEGIES" "RIGH FRON LEF BAC"
←GO-THROUGH /STRATEGIES/ AND /NOSE/ AND /TAIL/ AND "MAZE-3"
AT 2 4        (RIGH)
AT 1 4          "
AT 1 3          "
AT 2 3          "
AT 2 4          "
(BREAK)
I WAS AT LINE 20 IN GO-THOUGH
←
```

There is, however, a very simple strategy that will eventually unravel *any* maze! Random motion through a maze will pass over every square, though for a large maze it may take a very, very long time. With a few changes in NEXT, we can use the procedure RANDO (RANDOM is a built-in LOGO procedure) that moves the mouse randomly just as we used the basic four move procedures.

```
TO RANDO /NOSE/ AND /TAIL/
10 OUTPUT EXECUTE-2 OF
    (RANDOM-CHOOSE OF "FRON LEF RIGH BAC") AND
    /NOSE/ AND /TAIL/
END
```

RANDO uses the subprocedure EXECUTE-2 to actually compute the new position. It also calls RANDOM-CHOOSE to randomly choose one of the four possible move directions.

```
TO RANDOM-CHOOSE /LIST/
10 MAKE "RANDOM" SUM OF RANDOM AND 1  (Get a random number between
                                       1 and 10)
20 TEST GREATERP                      (Is our number bigger than
    /RANDOM/ AND                       the length of our list?)
    COUNT OF /LIST/
30 IF TRUE GO TO LINE 10              (If so, get a new one)
40 OUTPUT NTH OF /RANDOM/ AND /LIST/  (If not, use NTH to get the
END                                    proper element in the list)
```

Since it makes no tests on its choices, RANDO will sometimes output illegal positions. NEXT is equipped to catch these, as it did for our previous strategy lists. But, if /STRATEGIES/ is simply "RANDO", then NEXT, upon finding an illegal name, will go to try BUTFIRST of its strategy list. This new list is empty, so NEXT will declare itself STUCK. What we really want it to do is *keep trying* RANDO until it finally comes up with a legal next position. This can be accomplished by rewriting Line 70 of NEXT so that the failing move is put on the end of the strategy list instead of simply being discarded.

```
70 OUTPUT NEXT OF (SENTENCE OF BUTFIRST OF /STRATEGIES/
    AND FIRST OF /STRATEGIES/) AND /NOSE/ AND /TAIL/ AND
    /MAZE-NAME/
```

With this change, /STRATEGIES/ can be considered an endless loop that repeats itself periodically. The only problem with this is that, if we are truly STUCK, NEXT will get caught in an endless loop, forever retrying strategies that can never produce a legitimate move. We can take care of this possibility for some cases by specifying that any strategy list with "END" as its last element will not repeat. For example, a bear-left strategy that does not permit backwards motion can then be written "LEF FRON RIGH END". We would then get STUCK only by wandering into a cul-de-sac. To include this "END" modification in NEXT, we need only change our termination line in NEXT, Line 10, to read:

```
10 TEST IS FIRST OF /STRATEGIES/ "END"
```

We are now ready to experiment with our "random" strategy. We first want to use RANDO as our entire strategy list. This runs into a minor difficulty, however; NEXT expects /STRATEGIES/ to be a *sentence* and when instead it finds a *word* (RANDO), FIRST OF /STRATEGIES/ is taken to be the first letter of RANDO and an error message results. We need an initialization procedure that makes /STRATEGIES/ into a sentence (even if it is a one-word sentence) before we begin GO-THROUGH. The procedure MAZE sets up a use of GO-THROUGH by requesting the appropriate inputs and then, in its Line 50, turns any single word strategy into a sentence containing that single word.

```
TO MAZE
10 REQ "MAZE-NAME"
20 REQ "NOSE"
30 REQ "TAIL"
40 REQ "STRATEGIES"
50 MAKE "STRATEGIES" SENTENCE OF /STRATEGIES/ AND /EMPTY/
60 GO-THROUGH /STRATEGIES/ AND /NOSE/ AND /TAIL/ AND /MAZE-NAME/
END
```

REQ is a convenient little subprocedure for requesting and storing the thing of some name given as an input.

```
TO REQ /NAME/
10 TYPE WORD OF /NAME/ AND "?..."
20 MAKE /NAME/ REQUEST
END
```

The use of MAZE, and thus REQ, is shown next as we begin to try out RANDO.

```
←SHOW-MAZE "MAZE-1"
00000
++0+0
F00+0
++0+0
000+0
←MAZE
MAZE-NAME?...MAZE-1
NOSE?...2 1
TAIL?...1 1
STRATEGIES?...RANDO
AT 3 1
AT 2 1
AT 3 1
AT 4 1
AT 3 1
AT 3 2
AT 3 3
AT 2 3
I AM AT 1 3 AND I FOUND THE CHEESE!
←
```

Although this is an actual run, the speed with which it found the cheese was not typical. The subsequent run, for example, was interrupted after 62 moves. It would have succeeded eventually but we grew impatient.

Thus, we see two features of our random-motion strategy which are somewhat unpleasant. First, although it will always succeed, this method is not systematic. Also, we cannot put a bound on the number of moves it will take to succeed. Even for so simple a maze as the one below, the mouse will require more than 20 moves to find the cheese, about once in every thousand trials (because each time the mouse returns to the center square it has a fifty percent chance of finishing*).

```
+++++
+00F+
+++++
/NOSE/ IS "2 1"
/TAIL/ IS "1 1"
```

Of course, the mouse will find the cheese in one move in half of the runs.

The purely random strategy could have solved even the great Labyrinth, but the walls would surely have collapsed before the mouse stumbled onto the exit. However, with combinations of FRON, LEF, RIGH, BAC, RANDO can be used to make more efficient strategies that still assure eventual success in any maze. For example, the strategy "RANDO LEF FRON RIGH BAC" is essentially a bear-left, but the addition of an occasional random move is enough to ensure success with those mazes (like MAZES 2 and 3) where a simple bear-left would fail.

Students can learn a great deal by experimenting with composite strategies, with the goals of improving efficiency and generality (i.e., solving more types of mazes). For instance, a student may find that RANDO, when used in combination with other strategies, about-faces too much for his taste (BAC, after all, is the only move that will always be legitimate). This situation can be improved by editing RANDO to either eliminate BAC as a possibility, or to increase the odds against BAC by changing the list of procedures in Line 10 to, say, "FRON FRON LEF LEF RIGH RIGH BAC".

---

*By the same argument used for this case, we see, generally, the mouse will require more than 2n moves, once in every 2^n trials. (To convert to decimal note that $2^{10} \sim 1000$)

He may find that when used in combination with a bear-right strategy, he wants a RANDO weighted slightly in LEF's favor, and vice versa for bear-left and RIGH. He may find it useful to use two or three modified RANDOs mixed in with a fixed strategy.

There will, however, always be a built-in inefficiency in any strategy that includes RANDO. RANDO strategies are subject to this limitation -- no matter how sophisticated, they are necessarily hit-or-miss affairs. We next consider a new means of solving non-connected mazes that does not depend on RANDO.

3.6 Marking the Path

A hint taken from Theseus can point us towards some new techniques. Theseus escaped from the Labyrinth by marking his path (with a thread) and retracing his steps. Although we are not interested in retracing *our* steps, we are very much interested in knowing where we have been. Marking our path will help us avoid repeating unfruitful sequences. Instead of using a thread (or bread crumbs, like Hansel and Gretel), we can develop a neater way of marking, as follows. *All* legal squares initially contain 0. We can increment their content by 1 at each passing.

We will write a procedure, MARK, to add 1 to the contents of a given square. MARK resembles GET-STATUS in that it must locate a square within a maze-sentence. It differs significantly from GET-STATUS, however, in that it must reassemble the maze-sentence, since it redefines the maze by marking the specified square. GET-STATUS ignores the rest of the maze. Let's start from the bottom and define a procedure MARK-X that, given the proper row, /ROW/, and a column number, /X/, outputs the same row with the indicated square increased by one.

```
TO MARK-X /X/ AND /ROW/
10 TEST IS /X/ "1"              (Do we want to mark the first square?)
20 IF TRUE OUTPUT WORD OF
    SUM OF FIRST OF /ROW/ AND 1 (Yes, change the first square and
    AND BUTFIRST OF /ROW/        output the marked word)
30 OUTPUT WORD OF (FIRST OF /ROW/)   (Otherwise, leave the first
    AND MARK-X OF (DIFFERENCE OF      square unchanged and repeat
    /X/ AND 1) AND (BUTFIRST OF /ROW/) the process with the remain-
END                                   ing squares)
```

Thus,

```
←PRINT MARK-X 3 "+0001+"
+0101+
←
```

The next thing to write is a procedure MARK-ROW that locates the proper row to input to MARK-X. Just as MARK-X reassembles /ROW/, MARK-ROW must reassemble the sentence /MAZE/ as it goes along:

```
TO MARK-ROW /X/ AND /Y/ AND /MAZE/
10 TEST IS /Y/ "1"                (Do we want the first word?)
20 IF TRUE OUTPUT SENTENCE OF       (If so, send it to MARK-X
    MARK-X OF /X/ AND (FIRST OF     and append the result onto
    /MAZE/) AND (BUTFIRST OF /MAZE/) the rest of the sentence)
30 OUTPUT SENTENCE OF (FIRST OF     (Otherwise, peel one word off
    /MAZE/) AND MARK-ROW OF /X/ AND /MAZE/, diminish /Y/, and
    (DIFFERENCE OF /Y/ AND 1) AND   start again, meanwhile build-
    (BUTFIRST OF /MAZE/)            ing a sentence of the peelings)
END
```

We denote a square, elsewhere, as a single sentence composed of two numbers - row and column - rather than by two separate numbers. The procedure MARK modifies MARK-ROW to make it compatible with this convention.

```
TO MARK /POSITION/ AND /MAZE-NAME/
10 MAKE /MAZE-NAME/ MARK-ROW OF
    FIRST OF /POSITION/          (The column number)
    AND LAST OF /POSITION/       (The row number)
    AND THING OF /MAZE-NAME/     (The maze-sentence itself)
END
```

Let's try out MARK.

```
←SHOW-MAZE "MAZE-4"
000
+F+
←
←MARK "1 1" "MAZE-4"
←SHOW-MAZE "MAZE-4"
100
+F+
←
```

To incorporate this feature in our main system, we should mark the starting positions (both /NOSE/ and /TAIL/) and each subsequent position as it is reached. We can mark /NOSE/ and /TAIL/ by adding these two lines to MAZE:

```
54 MARK /NOSE/ AND /MAZE-NAME/
56 MARK /TAIL/ AND /MAZE-NAME/
```

The marking of subsequent positions is carried out in GO-THROUGH by adding the following line:

```
75 MARK /NOSE/ AND /MAZE-NAME/
```

Let us see how the whole thing works now:

```
←SHOW-MAZE "MAZE-2"
000+
0+0F
000+
←MAZE
MAZE-NAME?...MAZE-2
NOSE?...1 1
TAIL?...2 1
STRATEGIES?...RIGH FRON LEF END
AT 1 2
AT 1 3
AT 2 3
AT 3 3
AT 3 2
I AM AT 4 2 AND I FOUND THE CHEESE!
```

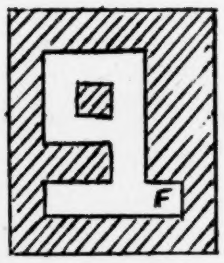
```
←SHOW-MAZE "MAZE-2"
110+
1+1F
111+
←
```

Since the maze is actually changing after each move in a GO-THROUGH, we might like to see it at intermediate moves, rather than just at the end. The display procedure, SHOW-MAZE, already exists, so all we have to do is put a request and a test in GO-THROUGH:

```
78 TEST IS REQUEST "M"          ("M" will be a signal announcing
                                that we want the maze shown)
79 IF TRUE SHOW-MAZE /MAZE-NAME/
80 IF FALSE PRINT SENTENCE OF "AT" AND /NOSE/
```

We can now get a much more dynamic view of a run through a maze:

```
←MAZE                                (MAZE-5 is  000+
MAZE-NAME?...MAZE-5                              0+0+
NOSE?...2 5                                      000+
TAIL?...1 5                                      ++0+
STRATEGIES?...LEF FRON RIGH BAC                  000F )
⁙M
000+
0+0+
000+
++0+
111F
```



```
⁙          (user hits carriage-return to indicate he
            does not want the maze displayed)
AT 3 4
⁙             ( " )
AT 3 3
⁙
AT 2 3
⁙
AT 1 3
⁙
AT 1 2
⁙
AT 1 1
⁙
```

87

```
AT 2 2
∷M          (This causes the MAZE to be displayed)
111+
1+0+
111+
++1+
111F
∷
AT 3 2
∷
AT 3 3
∷
AT 3 4
∷M
111+
1+1+
112+
++2+
112F
I AM AT 4 5 AND I FOUND THE CHEESE!
←
```

3.7 <u>History-dependent Strategies</u>

Now that we can see how often we've visited a particular square, let's try to use this information to develop better strategies. One way we might use it is to see if any of the legitimate moves will get us into a square which has not been previously explored. We call a square virgin if it has a status of 0 or "F". If we add to our basic moves a procedure which moves only into a virgin square, we can solve mazes like MAZE-2 or MAZE-3 without counting on RANDO to get our mouse out of loops. Let's write such a procedure and call it VIRGIN.

VIRGIN makes trial moves forward, left, and right (BAC can never yield a virgin square, since it was just occupied on the previous move) and output the position of the first square it finds with a status of 0 or "F". If no such squares are adjacent, VIRGIN outputs /EMPTY/ to indicate failure. There is never any ambiguity since /EMPTY/ can never denote the position of a square.

VIRGIN calls a subprocedure, VIRGIN-2, with a list of moves to be tried.

```
TO VIRGIN /NOSE/ AND /TAIL/
10 OUTPUT VIRGIN-2 OF /NOSE/ AND /TAIL/ AND
    "LEF FRON RIGH"
END
```

```
TO VIRGIN-2 /NOSE/ AND /TAIL/ AND /PROCEDURES/
10 TEST EMPTYP /PROCEDURES/      (Have we exhausted all possibilities?)
20 IF TRUE OUTPUT /EMPTY/        (If so, indicate failure)
30 MAKE "TRIAL" EXECUTE-2 OF     (Compute trial square resulting
    (FIRST OF /PROCEDURES/)      from first move)
    AND /NOSE/ AND /TAIL/
40 MAKE "STATUS" GET-STATUS OF   (Compute its status)
    /TRIAL/ AND /MAZE-NAME/
50 TEST EITHER (IS /STATUS/ "F")
    (IS /STATUS/ 0)              (Is it virgin?)
60 IF TRUE OUTPUT /TRIAL/        (If so, output the square)
70 OUTPUT VIRGIN-2 OF            (If not, keep looking)
    /NOSE/ AND /TAIL/ AND
    BUTFIRST OF /PROCEDURES/
END
```

Note that this version of VIRGIN is one of six that can be written, each with a different order of LEF, FRON, and RIGH in /PROCEDURES/. When there are two or more virgin possibilities, these different VIRGINS will give different results, since the order in which they make their trial moves determines which square is chosen. The VIRGIN we have written above tries moves in the order LEF, FRON, then RIGH, and thus, if given a choice, will bear left.

Notice also that if VIRGIN does not find an unmarked square, it will output /EMPTY/. Before we can use VIRGIN as part of a strategy, we will have to change NEXT slightly to allow for this /EMPTY/ possibility. We can do this by checking whether /TRIAL/ is equal to /EMPTY/. If so, we go on to try the next move. We accomplish this by adding two lines to NEXT:

```
34 TEST EMPTYP /TRIAL/
36 IF TRUE GO TO LINE 70
```

To illustrate the effectiveness of VIRGIN, let's try this new basic strategy on MAZE-3.

```
←MAZE
MAZE-NAME?...MAZE-3
NOSE?...2 1
TAIL?...1 1
STRATEGIES?...VIRGIN
∷M
11100
00000
00F00
00000
00000
∷
AT 4 1
∷
AT 5 1
∷
AT 5 2
∷
AT 5 3
∷
AT 5 4
∷
AT 5 5
∷
AT 4 5
∷
AT 3 5
∷M
11111
00001
00F01
00001
01111
∷
AT 1 5
∷
AT 1 4
∷
AT 1 3
∷
AT 1 2
∷M
11111
11001
10F01
10001
11111
∷
AT 3 2
∷
AT 4 2
∷
AT 4 3
∷
AT 4 4
∷
AT 3 4
∷
AT 2 4
∷M
11111
11111
11F11
11111
11111
I AM AT 3 3 AND I FOUND THE CHEESE!
←
```

Although, in many ways VIRGIN is a powerful procedure, it is extremely susceptible to cul-de-sac problems. When the mouse reaches the end of a cul-de-sac, VIRGIN can give no indication of how to retrace its steps. Fortunately, however, we know how to take care of such dead ends once and for all. Any square that permits only one possible move out of it must be the end of an alley. Once we have identified such a square, we can replace its contents with a "+". Subsequently then, this dead-end square will appear as a wall. This process can be repeated recursively until the entire cul-de-sac is closed off. The resulting maze is therefore simpler and may become tractable to strategies which would otherwise fail.

So, we will write a procedure, CUL-DE-SAC, to identify dead-end squares and close them off. If the square currently occupied is not a cul-de-sac, CUL-DE-SAC will output /EMPTY/; if it is, CUL-DE-SAC will output the one permissible move - backing out - and mark the dead-end square with a "+". To do this we will have to modify MARK to accept three inputs, /POSITION/, /MAZE-NAME/, and /CHARACTER/ because we will have to specify the /CHARACTER/ to be marked.

The first thing we will need is a procedure TEST-WALLS to find whether or not a given square is a cul-de-sac. TEST-WALLS tries the three basic moves FRON, LEF, RIGH (BAC is excluded because it is always a possible move) and outputs "TRUE" if all of them are blocked (i.e., the given square is a cul-de-sac) or "FALSE" otherwise.

```
TO TEST-WALLS /NOSE/ AND /TAIL/ AND /PROCEDURES/
10 TEST EMPTYP OF /PROCEDURES/  (Have we exhausted the possibilities?)
20 IF TRUE OUTPUT "TRUE"        (If so, we have a cul-de-sac)
30 MAKE "TRIAL" EXECUTE-2 OF
     FIRST OF /PROCEDURES/ AND
     /NOSE/ AND /TAIL/
40 MAKE "STATUS" GET-STATUS OF
     /TRIAL/ AND /MAZE-NAME/
50 TEST (EITHER IS /STATUS/ "+")    (Is it a wall?)
     (EMPTYP /STATUS/)
60 IF TRUE OUTPUT TEST-WALLS OF     (If so, try the remaining
     /NOSE/ AND /TAIL/ AND           possibilities)
     (BUTFIRST OF /PROCEDURES/)
```

```
70 OUTPUT "FALSE"                 (There is at least one way out,
END                               other than BAC, so we don't have
                                  a CUL-DE-SAC)
```

CUL-DE-SAC now follows easily. There is just one tricky point, implemented in step 30. Up to now, the tail of the mouse is put at the previous position of the nose (/OLD-NOSE/ in GO-THROUGH). Here, however, in CUL-DE-SAC, if TEST-WALLS succeeds, /NOSE/ is marked as a wall. In this case we want the mouse to back out. This is easily done by noting that the square "behind" the tail of the mouse (/NOSE/ /TAIL/) is in front of the mouse given by /TAIL/ /NOSE/. And we can use NEXT to do this.

```
TO CUL-DE-SAC /NOSE/ AND /TAIL/
10 TEST (TEST-WALLS OF /NOSE/       (Are we in a dead-end square?)
   AND /TAIL/ AND "FRON LEF RIGH")
20 IF TRUE MARK /NOSE/ AND          (If so, make it a wall)
   /MAZE-NAME/ AND "+"
30 MAKE
   "OLD-NOSE"
   NEXT OF ("LEF FRON RIGH")
   /TAIL/ /NOSE/ /MAZE-NAME/
40 IF TRUE OUTPUT BAC               (And output the one possible
   /NOSE/ AND /TAIL/                 move, BAC)
50 OUTPUT /EMPTY/                   (Otherwise, indicate failure)
END
```

We still have to rewrite the MARKing procedures to handle the additional input, /CHARACTER/. When MARK is to be used as originally intended, to increase the status of a square by 1, it will be called with /CHARACTER/ set to /EMPTY/. The main change required is the incorporation of a test in MARK-X. The only other changes involve the addition of the new input to the title lines of the various procedures. The rewritten MARK package looks like this: (the additions are all underscored)

```
TO MARK /POSITION/ AND /MAZE-NAME/ AND /CHARACTER/
10 MAKE /MAZE-NAME/ MARK-ROW OF FIRST OF /POSITION/
    AND LAST OF /POSITION/ AND THING OF /MAZE-NAME/
    AND /CHARACTER/
END
```

```
TO MARK-ROW /X/ AND /Y/ AND /MAZE/ AND /CHARACTER/
10 TEST IS /Y/ 1
20 IF TRUE OUTPUT SENTENCE OF MARK-X OF /X/ AND FIRST OF
    /MAZE/ AND /CHARACTER/ AND BUTFIRST OF /MAZE/
30 OUTPUT SENTENCE OF FIRST OF /MAZE/ AND MARK-ROW OF /X/
    AND DIFFERENCE OF /Y/ AND 1 AND BUTFIRST OF /MAZE/
    AND /CHARACTER/
END
```

```
TO MARK-X /X/ AND /ROW/ AND /CHARACTER/
10 TEST IS /X/ 1
20 IF FALSE OUTPUT WORD OF FIRST OF /ROW/ AND MARK-X
    OF DIFF OF /X/ AND 1 AND BUTFIRST OF /ROW/
    AND /CHARACTER/
30 TEST EMPTYP /CHARACTER/        (Are we replacing or adding 1?)
40 IF TRUE OUTPUT WORD OF SUM OF
    FIRST OF /ROW/ AND 1 AND
    BUTFIRST OF /ROW/             (Adding 1)
50 OUTPUT WORD OF /CHARACTER/ AND
    BUTFIRST OF /ROW/             (Replacing)
END
```

We must not forget, finally, to bring the other references to MARK up to date. In MAZE, the two lines that mark the starting position should be rewritten -

```
54 MARK /NOSE/ AND /MAZE-NAME/ AND /EMPTY/
56 MARK /TAIL/ AND /MAZE-NAME/ AND /EMPTY/
```

So must the line in GO-THROUGH that marks each subsequent position:

```
75 MARK /NOSE/ AND /MAZE-NAME/ AND /EMPTY/
```

Now let's try CUL-DE-SAC, combined with VIRGIN, on MAZE-1.

```
←MAZE
MAZE-NAME?...MAZE-1
NOSE?...1 1
TAIL?...1 2
STRATEGIES?...CUL-DE-SAC VIRGIN END
∷M
+2000
++0+0
F00+0
++0+0
000+0
∷
AT 3 1
∷
AT 4 1
∷
AT 5 1
∷
AT 5 2
∷
AT 5 3
∷
AT 5 4
∷
AT 5 5
∷M
++111
++0+1
F00+1
++0+2
000++
∷
AT 5 3
∷
AT 5 2
∷
AT 5 1
```

```
∷
AT 4 1
∷
AT 3 1
∷M
+++++
++1++
F00++
++0++
000++
∷
AT 3 3
∷
AT 3 4
∷
AT 3 5
∷
AT 2 5
∷
AT 1 5
∷
AT 2 5
∷
AT 3 5
∷
AT 3 4
∷
AT 3 3
∷M
+++++
+++++
F1+++
+++++
+++++
I AM AT 1 3 AND I FOUND THE
CHEESE!
←
```

The strategy "LEF FRON RIGH BAC" forces the mouse to cling to the left-hand wall. Clinging to the left-hand wall is not a bad thing, as long as all paths in a maze are one square wide. But, as we saw in the case of MAZE-3, any open areas will only be skirted and their central regions ignored by such a strategy. Covering open areas, however, is the special strength of VIRGIN. For example, VIRGIN will, by itself, simply spiral in on an open space, covering all possibilities, as shown in Figure 2.

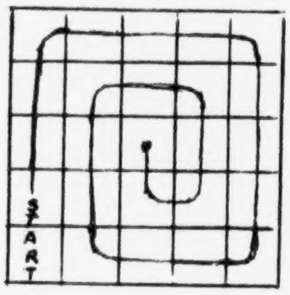


Figure 2.

Another class of mazes which we can systematically solve using VIRGIN with the sequence "LEF FRON RIGH BAC" is that for which the bear-left strategy used by itself will miss a single path. For example:

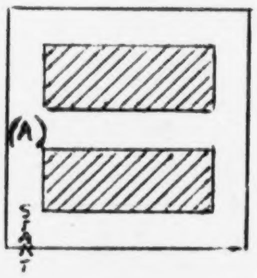
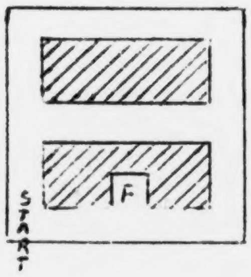


Figure 3.

"LEF FRON RIGH BAC" alone will loop around the outside corridor indefinitely. "VIRGIN LEF FRON RIGH BAC" will do the same the first time around. The second time it reaches point (A), however, VIRGIN will force the mouse down the center corridor. Thus, the use of VIRGIN, as above, usually gives one extra path when the maze is more "complicated" than a simple ring. Also, in most cases, VIRGIN will systematically cover open areas within a more complicated maze.

We still have the difficulty, however, that once a given *part* of a maze is covered once -- its entrance and exit having been marked -- VIRGIN will never get us back for another try. For example, the strategy "VIRGIN LEF FRON RIGH BAC" applied to the maze in Figure 3 results in the outside corridor being completely traced, then causes indefinite looping around the upper block. If we modify this maze, as in Figure 4, we have a maze we cannot solve using that strategy. Once the basic figure-eight pattern is traversed, the lower corridor is never retraced and VIRGIN never is able to find F. Also, since there are no dead-end squares other than F, the addition of CUL-DE-SAC in any manner is of no help.



or

```
00000
0+++0
00000
0+++0
0+F+0
00000
```

Figure 4.

VIRGIN will, in sufficiently complex mazes, force the mouse to take at least one new path; one which it would not otherwise have taken. But, at least in some cases, as the one above, VIRGIN will force precisely one such new path and one is not always sufficient. By extending VIRGIN, we can design a strategy which successively forces all paths to be tried. The strategy was suggested in a poem by Robert Frost, who, confronted by "two roads diverged in a yellow wood," after some deliberation, "took the one less traveled on, and that has made all the difference." This turns out to be an excellent principle for us as well, for it will enable us to cover all squares in any maze.

This universal strategy entails taking the least-traveled path at any node, not just an untraveled (unmarked) one, as VIRGIN does. It is easy to see that this strategy always works, by an argument leading to a contradiction. Assume that this "least-marked" strategy leads the mouse into an endless loop. The numbers of the squares involved in such a loop must then increase without limit. The loop clearly does not involve every square in the maze because, if it did, the finish would have been found. Also, there must be some path from the loop to the finish. One square on this path must be adjacent to the loop but not in it. But such a square cannot exist, since eventually our strategy would force this square to be traversed. Thus we have a contradiction, and we see our strategy indeed covers the entire maze unless a finish is found.

How do we write a procedure for this strategy? Basically, we will want it to try all four standard moves, examine the status of the four resulting squares, and output the legal move which has the lowest number associated with it. The structure of this program will be very much like VIRGIN, with one major change. We will now have to keep a running "BEST" move, which is the best trial so far. /BEST/ will start off as BAC OF /NOSE/ AND /TAIL/, since that is always a legal move. We will then compare the results of LEF, FRON, and RIGH with it - in each case retaining the best square. We will write an initialization procedure, ROBERT-FROST, a status comparison procedure, MOREP, and a procedure, LEAST-TRAVELED-ON, to perform most of the work.

```
TO LEAST-TRAVELED-ON /NOSE/ /TAIL/ /MOVES/
10 TEST EMPTYP /MOVES/          (Are there any more moves to be
                                 tried?)
20 IF TRUE STOP
30 MAKE
    "TRIAL"                     (If so, make "TRIAL" the first
    EXECUTE-2 OF FIRST /MOVES/   of them)
     AND /NOSE/ AND /TAIL/
40 MAKE
    "TRIAL STATUS"              (Find the status of /TRIAL/)
    GET-STATUS OF /TRIAL/ AND
     /MAZE-NAME/
50 TEST MOREP                   (If it is equal to or less than
    /TRIAL STATUS/              the number in /BEST/)
    GET-STATUS OF /BEST/ AND
     /MAZE-NAME/
60 IF FALSE MAKE "BEST" /TRIAL/ (We have a new /BEST/)
70 LEAST-TRAVELED-ON /NOSE/ /TAIL/  (Repeat for rest of set of
    BUTFIRST /MOVES/                 moves)
END
```

Here is the subprocedure MOREP:

```
TO MOREP /TRIAL/ /NUMBER/
10 TEST NUMBERP /TRIAL/         (Is /TRIAL/ a number?)
20 IF TRUE OUTPUT GREATERP      (We know the second input to
    /TRIAL/ /NUMBER/            MOREP is always a number.  If the
30 OUTPUT "TRUE"                first input is also a number, we
END                             compare these with GREATERP, if
                                not we output "TRUE" so that
                                Line 60 in LEAST-TRAVELED-ON
                                fails)
```

Robert-Frost itself is very simple:

```
TO ROBERT-FROST /NOSE/ /TAIL/
10 MAKE "BEST" BAC /NOSE/ /TAIL/
20 LEAST-TRAVELED-ON /NOSE/ /TAIL/ "LEF FRON RIGH"
30 OUTPUT /BEST/
END
```

As with VIRGIN, there are actually a number (twenty-four, in this case) of possible ROBERT-FROST's, depending on the order in which the basic strategies are tested. The one we have written bears left when given a choice between two equivalent moves. It clearly works for the maze given in Figure 4, the simplest maze for which VIRGIN and bearing left fails. We also show how ROBERT-FROST works for a more complicated maze. Rather than go through a run step-by-step, however, we will simply indicate the path it would take by drawing a line through the maze.

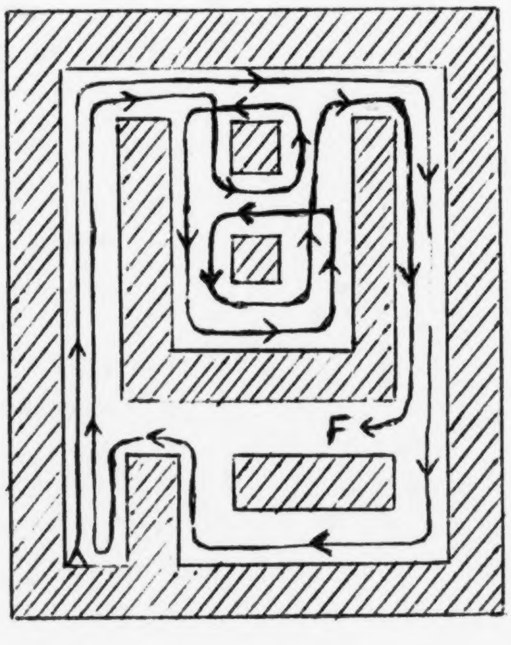


Figure 5.

At this point we are by no means near to exhausting the possibilities of developing maze strategies. There are several directions for elaborating and deepening the strategies thus far studied -- some of these are considered in the associated set of problems. For example, we consider methods for using the status history to simplify the maze by eliminating higher order loops than those found using CUL-DE-SAC. Also, we use the status history to investigate the problem of finding minimal paths. Other problems involve significant extensions such as generalizing to three-dimensional (and higher) mazes and developing more global strategies, based upon entire paths instead of squares.

These problems merely indicate some obvious continuations of the development thus far made. Indeed, the context of maze strategies is mathematically and pedagogically very rich and can be the basis for a considerably more elaborate course than is possible as part of a single-unit high-school sequence.

PROBLEMS

LOGO Sequence on Strategy in Problem-Solving

1. NUMBER GUESSING PROBLEMS

1. Write a series of procedures to play the following game: The user and the computer take turns guessing each other's numbers. The winner of each round is the player who guesses the other's number first, and the winner of the game is the player who wins the most rounds. The computer should use a binary search, perturbed slightly to mask its strategy from the user. It might, for example, instead of guessing the midpoint of the interval precisely, modify this by adding or subtracting a random number within ten percent of the midpoint.

2. The computer can improve its play by dishonest, underhanded means. To do this, modify the computer's number-choosing procedure as follows. Instead of choosing a number in advance of play, the computer will put off its choice to require the maximum number of guesses from its opponent. At each guess, this procedure, call it SHAGGY-DOG, should respond HIGH or LOW depending on which choice yields the larger remaining interval. Only when the interval has been reduced to a single number does SHAGGY-DOG reveal "its" choice. What is the best counter-strategy for the player?

3. (A) Extend the number guessing procedures for use in a two-dimensional guessing game. The user will try to guess a *pair* of numbers within a given range. The computer compares the current guess with the previous one. It responds "HOT", if both numbers are closer than on the previous guess; "COLD", if both are further off than before; and "WARM", if one number is closer but the other further.

(B) Is there a "best" strategy for this pair-guessing game comparable to binary search for single number guessing? How would you compute the relative efficiency of strategies for pair-guessing?

(C) Extend this game by "teaching" the computer to guess a pair of numbers that the user has chosen.

2. Number-Sequence Extrapolation Problems

1. Erasing Procedures

(A) To eliminate procedures like the original DIV-TWO which generate errors and force OCCAM to a halt, write a procedure called CLOBBER, with a procedure name as input, which erases the procedure and removes it from the procedure banks.

(B) OCCAM's banks will eventually contain many, many redundant procedures, i.e., procedures which describe the same sequence. Write a procedure called CLEAN-UP which exercises the banks searching for apparently equivalent procedures. When it finds a pair of successor functions which *seem* to give the same sequence, it should use some criterion for choosing between them and use CLOBBER to erase one of them.

(C) Write two successor procedures which are taken to be equivalent by the redundancy criterion you have chosen, but which actually describe different sequences, at least in part.

2. Successive and Joint Generalization

The generalizer we have written replaces only the *first* constant it finds in a procedure. Sometimes, however, a more useful generalization may involve other constants, as well.

(A) Write a GENERALIZE that produces a generalized procedure for each constant in a given successor procedure. The first generalized procedure produced in this manner replaces the first constant; the second generalized procedure so produced leaves the first constant unchanged, but replaces the second constant; and so on.

(B) Discuss the advantages and disadvantages of writing a generalization procedure for simultaneously replacing all the constants contained in a procedure (that is, a procedure containing three constants would be generalized to a four-input procedure).

3. A Bug in GENERAL-TESTER?

There is a bug, or at least a weakness, in GENERAL-TESTER as given in the text. When GENERAL-TESTER finds a procedure and /DUMMY/ value that reproduce a given sequence, these are given to APPROVAL. If, however, the /NEXT-TERM/ generated by that particular procedure and /DUMMY/ does not satisfy APPROVAL, the next generalized *procedure* is invoked rather than the next value of /DUMMY/.

(A) How can GENERAL-TESTER be modified so that it runs through *all* eleven /DUMMY/ values for each procedure - unless the correct /NEXT-TERM/ is found?

(B) Make up an example of a sequence and a procedure for which two different values of /DUMMY/ will satisfy GENERAL-TESTER?

4. Multiple Answers

Many sequences look identical over a limited number of terms. Even after a next term has been approved, the extended sequence might be described by a number of different successor functions. Modify OCCAM so that, if the user requests it to do so, it will seek out *all* the procedures that give a successful extrapolation, not just the first one that it finds.

5. Reordering Procedure Banks

Some successor functions are a great deal more useful than others; a generalized ADD-ONE, for example, will "win" more often than, say, NEXT-FACTORIAL. OCCAM's overall efficiency will depend on the order in which it tests the contents of its banks. We can make OCCAM faster by having it test the most effective successor functions first, and then go on to the less used ones.

(A) There are at least two ways in which OCCAM can be modified to reorder its two procedure banks. A straightforward one is to move the winning procedure to the beginning of the appropriate bank (generalized or ungeneralized) at the end of each run. Using this method, the most popular procedures will stay near the beginning of the banks while the less used ones will drift to the ends. Write such a REORDER procedure.

(B) Another method is to keep track of the number of times each successor function "wins". Using these scores, OCCAM can try procedures in order of their previous success. This method is slower than the other and involves more manipulation of the procedure lists. However, it is likely to require fewer tries on a run and it develops statistics on the operational utility of procedures. What specific procedures are needed to do this reordering? List all of these, specifying their operations and how they relate to each other.

6. Other Sequence Transformations

Generating a difference sequence is only one of many transformations we can use to help solve sequence problems.

(A) Sometimes dropping the first term will permit OCCAM to solve a sequence it could not otherwise extrapolate. Sometimes the sequence made up of every other term will be tractable, where the original sequence was not. Both these transformations can be easily incorporated into OCCAM. In these cases (unlike difference sequences), the next term of the transformed sequence is also the next term of the original. Thus, we can simply send the transformed sequence back to the old list scanners. Incorporate these two strategies in OCCAM.

(B) Other useful sequence transformations can be obtained by taking partial sums, products, or quotients. Each of these transformations will require its own rule for finding the next term of the original sequence and, consequently, a new set of list scanners to augment SCAN-DIFF and SCAN-GEN-DIFF. Modify OCCAM to try these transformation rules if its other strategies are unsuccessful.

7. Successive Differences

If we generate a difference sequence which we are unable to extrapolate, we can treat it as a new problem and study *its* difference sequence. For example, neither the sequence

a) 1 2 4 8 15 26

nor its difference sequence

b) 1 2 4 7 11

have an apparent solution. However, the second order difference sequence, which is (b)'s difference sequence is

c) 1 2 3 4.

This is a familiar arithmetic sequence whose successor is taken as 5; (b)'s next term will be computed as 16 and the extrapolation for the original problem will be taken as 42.

In general, the process of taking successive differences can be used to extrapolate any finite sequence. For example, consider this sequence and its successive DIFF-SEQUENCES:

```
4 3 5 15
-1 2 10
3 8
5
```

Though none of these sequences are obviously "natural" like the arithmetic sequence above, the last two -- "3 8" and "5" -- are easily extended. Let's choose the last one and assume that a one-term sequence is the beginning of a constant sequence. By this assumption, each sequence can be extended one term as follows:

```
4 3 5 15 38
-1 2 10 23
3 8 13
5 5
```

Write a procedure, EXTEND, that uses DIFF-SEQUENCE to recursively take differences of the original sequence until it is reduced to a one term. Use the assumption that this one-term sequence is constant to compute next-terms for each difference sequence and output, finally, the next term of the original problem. The value of EXTEND is that it always produces an answer, however contrived.

8. Number Sequence Quizzes

The procedure OCCAM takes a sequence from a user as input and (hopefully) outputs its next term.

(A) Design a procedure which reverses the role of user and computer by generating a sequence for the *user* to extrapolate.

Like OCCAM, this new procedure, SEQUENCE-QUIZ, needs a procedure-bank from which it can (randomly) choose a successor rule. It can also use a procedure like RUN to generate five or six terms, the last of which will be the extrapolation the user is to determine.

(B) Write a more sophisticated quiz to use strategies like differencing, partial sums, mixing sequences of alternate terms, etc. to transform the original sequence and generate harder problems. (Transformation of transformation can generate *very* difficult problems.)

(C) Make the number and type of sequence transformations depend on the user's past performance. For example, SEQUENCE-QUIZ might vary the difficulty of its problems to keep the user's success ratio at about 50%.

(D) How would you change your program to *teach* sequence-solving rather than just to test it?

9. Function Guessing

OCCAM as it how stands could be interpreted as a very limited function guessing procedure. We could say that it recognizes functions on the natural numbers if we take a sequence to mean the element of the range corresponding to the successive integers 1, 2, 3 ... . We can considerably improve this function guessing ability. Problems could be given to OCCAM in the form of a list of pairs corresponding to the unknown function, followed by a single input. OCCAM would then determine the output associated with this unpaired input. For example:

```
2,4
5,25
7,49
4,?...
```

As before, the new OCCAM seeks among its stored procedures for one which not only reproduces the given pairs, but acceptably completes the final pair.

(A) Make OCCAM into such a function-guesser.

(B) Extend this new function guessing OCCAM to handle functions on letters as well as numbers, for example, the alphabetic successor function:

```
A,B
B,C
C,D
⋮ ⋮ ⋮
```

or the alphabetic indexing function:

```
A,1
B,2
C,3
⋮ ⋮ ⋮
```

and similar "natural" rules.

3. MAZE PROBLEMS AND PROJECTS

3.1

In the text, a procedure TEST-WALLS is given which determines whether a given square is a dead-end. The concept embodied in TEST-WALLS is that if a square is a dead-end only the backwards move (BAC /NOSE/ /TAIL/) is allowed.

a) Write a procedure CORRIDORP /NOSE/ /TAIL/ which tests if a given square is a corridor--i.e. there are exactly two "legal" squares adjacent.

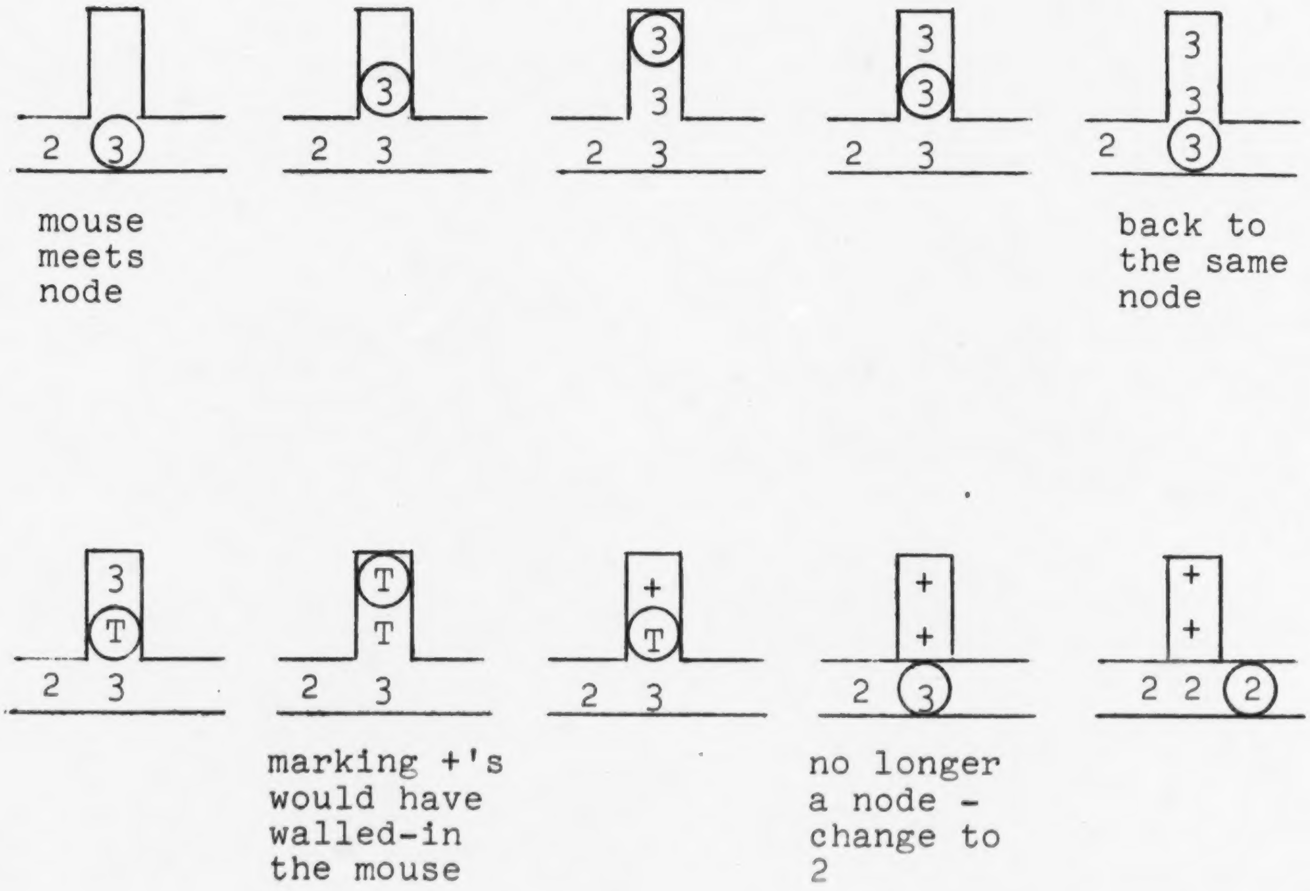
b) A square which has more than two "legal" adjacent squares is called a node. Nodes are important because they are the only squares at which a choice must be made. Write NODEP which tests a given square to determine whether or not it is a node.

3.2

The text gave a procedure, CUL-DE-SAC which blocked off dead-ends. We can use NODEP, similarly to eliminate simple loops--those on which no other nodes lie. To do this, we must modify our marking procedure. We mark squares, starting with the number 1, increasing the marking number only when a node is encountered If we encounter a node which has the same number as the path being marked, we have just traveled over a simple loop. We can eliminate this loop with a procedure RETRACE, which simple goes backwards, putting +'s in place of the current number, until the node is reached again. Or, better, we can mark the entire loop with some temporary symbol, say, T. Then once the whole loop has been thus marked, we replace all the T's with +'s.

This latter strategy also takes care of dead-ends as a special case, as we see below in Figure P1 and prevents the mouse from walling itself within such a cul-de-sac; the current nose position in each case is circled.

Figure P1.


2 3
2 3 3
2 3 3 3
2 3 3 3
2 3 3 3
2 3 3 T
2 3 T T
2 3 T +
2 3 + +
2 2 2 + +

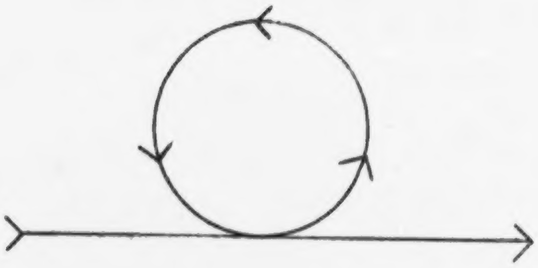
mouse meets node

back to the same node

marking +'s would have walled-in the mouse

no longer a node - change to 2

Figure P2.



A simple loop which does not contain F can never lead to anyplace interesting

3.3

There are 24 ways to order the four basic types of moves. Some of these are generally useful--bearing left for example, is given by LEF FRON RIGHT BAC, This covers at least one loop in any given maze.

FRON BAC RIGH LEF is much less useful--the mouse quickly gets caught in an oscillation. Give some rules to separate out the "good" strategies from the "bad" strategies.

3.4

Including RANDO in any strategy means that one cannot give an exact number corresponding to the moves it takes to solve a given maze. Nevertheless, an average figure can be given for a given maze and a given strategy corresponding to the time it would take averaged over a large number of trials.

a) Modify MAZE and GO-THROUGH so that
    (1) they count the number of moves made by the mouse
    (2) they do not indicate the current position of the mouse, until it "finishes"
    (3) the maze is not marked. [(2) and (3) help save time]

b) Try the strategy consisting solely of RANDO for a given maze, a large number of times. To keep the problem of reasonable size, it is best to use a simple maze in which the start and finish are separated by no more than five squares. Repeat your trials until you see a pattern in the number of times a given number of moves is required.

3.5

A novel mouse-maze strategy takes into account the very high reproduction rate of mice. One starts off with two mice, a male and female, who travel together. At any node they wait until the number of their progeny suffices to explore (again in pairs) all paths leading from the node. If a pair of mice hits a dead-end, they simply stop and wait there.

a) Show that at any node it suffices to send mice down only untraveled paths. Explain why this is the case whereas, for one mouse such a strategy does not suffice.

b) To implement this new strategy, it is probably best to completely rewrite NEXT, which gives the results of one move. One way to do this is to use a list of tails and noses corresponding to all existing mice. NEXT then will take the first nose and tail, and test whether they correspond to a dead-end. If so, we can simply output NEXT of the rest of the nose and tail list (i.e. forget about that mouse). If this fails, NEXT then will try CORRIDORP written for problem 2. If CORRIDORP is "TRUE" exactly one move is possible and the new nose and tail replace the old nose and tail on their respective lists. If CORRIDORP also fails, the current square must be a node. In this case, noses and tails corresponding to all untraveled paths are placed on the nose and tail lists. NEXT stops either when the lists /NOSES/ and /TAILS/ are empty or when it encounters a finish square. There is no longer any strategy list for NEXT since all possibilities are tried whenever a choice comes up.

GO-THROUGH needs no longer update the tail positions, since this is now done by NEXT. All GO-THROUGH need do is apply NEXT successively until NEXT outputs, say, "FINISH", indicating it is done.

3.6

a) An ambitious extension of the maze procedures given in the text is in their application to three-dimensional mazes. The major task is that of choosing and incorporating the more complex data structure for three-dimensional mazes. This phase includes ACCEPT-MAZE and SHOW-MAZE (How should a 3-D maze be shown on the teletype and stored in the computer?)

A square of the maze can now be indicated by three numbers rather than two, thus references to squares in GET-STATUS and MARK must be revised. Also two new moves UP and DOWN should be defined and the existing moves modified slightly to take the new coordinate triples into account. Small changes are also required in ROBERT-

FROST, CUL-DE-SAC and VIRGIN. (But not in MAZE, GO-THROUGH or NEXT!)

b) The generalization process described in part **a** can be extended still further to higher dimensions. In fact, the dimension of the maze can be left as an input parameter. The data structure must now be extensible in a systematic way, which presents some difficulties, but for describing positions, n-triples are not harder to work with than triples. It is extremely interesting, for this generalization to write a program which generates the programs corresponding to all possible moves from a given square.

3.7

A maze is certainly disconnected if the strategy LEF FRON RIGH BAC, corresponding to bearing left, returns the mouse to the starting position.

a) Modify MAZE so that the starting /NOSE/ is marked with an "S"

b) Modify GO-THROUGH so that a status of "S" stops the mouse and causes an appropriate message to be printed.

c) Determine whether returning to the start position means a maze is disconnected, regardless of strategy used.

3.8

It is very convenient at times for the mouse to have a memory. For example, a mouse can use its memory to help improve its path, once it has traced over some route successfully. The memory can, perhaps, best consist of the sequence of pairs corresponding to the squares it has visited. To create such a memory we can have MAZE initialize /MOUSE MEMORY/ to be the starting position and GO-THROUGH increase this list, each time a move is made.

To improve the path given as /MOUSE MEMORY/ at the end of a run, the mouse should successively look at each pair on the list. If any such pair recurs on the list we erase everything back to, and including, the previous cccurrence of that pair.

a) Show that repeatedly applying this procedure to /MOUSE MEMORY/ gives the shortest path contained within /MOUSE MEMORY/

b) Clearly, the improvement strategy described above does not give the shortest path on the list, as seen from Figure P3, where a bear left strategy was employed. What sort of strategy is best when the path found is to be optimized in the above manner?

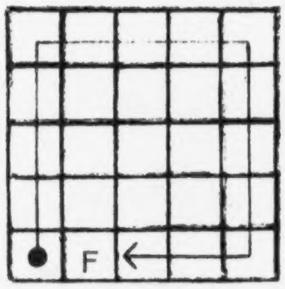


Figure P3

c) Implement the memory described above and describe the class of mazes for which it is most useful.

d) Only nodes really need to be remembered, since they are the only places where choices can be made. Modify the mouse memory to remember only nodes.

Volume 3, Part 2

STORY PROBLEMS IN ALGEBRA

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

George Lukas
Wallace Feurzeig

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

115

CONTENTS

Page

1. Introduction 1

2. Different Ways of Writing Equations 3

2.1 Infix Expressions and Prefix Expressions 3

2.2 Tree Diagrams 6

2.3 A Translation Problem 11

2.4 Extending the Infix-Prefix Translator 26

2.5 Evaluating Infix Expressions Automatically 38

2.6 An Infix-Prefix Translation Quiz 42

2.7 Completing the Infix-Prefix Translator 47

3. The Translation of Simple Story Problems: Infix Expressions to Story Expressions 58

3.1 Substitution 58

3.2 Simple Equations and Simple Story Problems 71

3.3 Rate Problems and Systems of Equations 81

Appendix A. TREE, A Tree Drawing Program

Appendix B. SEPARATE, An "Unpacking" Procedure

Appendix C. RND-CHOOSE, A Random Choice Procedure

Problems

LOGO and Story Problems
Teacher's Text

1. Introduction

This part of the LOGO mathematics course treats the representation and solution of algebra word or story problems. These problems, including the familiar age, rate, mixture, work, etc. stereotypes, are not important for their inherent mathematical interest. Their importance lies, rather, in that they lend themselves beautifully to showing the relation of formal methods to informal situations.

Algebra story problems are, for most beginning students, the first really "noisy" mathematical situations with which they are confronted. Actually, the standard story problems of school algebra are only of a few different types. Within each type of problem, though, the veneer of varied names, adjectival phrases, idioms, and the like, often hide the essential similarities from most students. Up to this point in their experience, they have not encountered such a variety of apparently "open" problems.

Thus, it is not surprising that story problems often are so very difficult for students -- not only to solve or even to intelligently cope with but sometimes even to approach. We have heard students put this comprehension gap rather nicely by saying about an unfamiliar type of story problem "but this is English, not math". Part of the difficulty certainly lies in our failure to provide students with a good grasp of formal concepts, such as the concept of formal *procedure*. The students' previous work with LOGO should be of help in this regard. Another way of helping is to develop sequences of translation procedures connecting formal problems with informal representations. The material in this unit is specifically directed in this way.

We begin by studying formal arithmetic operations. We then build up expressions in standard infix notation as, for example, X + 3. We gradually and systematically develop procedures for translating these infix expressions into equivalent prefix form (e.g., SUM OF X AND 3). We then develop procedures to translate these prefix expressions, in turn, into a variety of story expression forms (e.g., JOHN'S AGE 3 YEARS FROM NOW or 3 MILES FARTHER THAN YPSILANTI).

Finally, we construct comprehensive procedures for fully translating infix equations into a rich variety of story problems. Our object is not to develop a universal procedure for translating equations into arbitrary stories, but rather to develop *easy* and extensible procedures for handling a number of diverse story situations. The inverse translating, going from story problems to formal equations, is handled indirectly by writing quiz procedures which generate and then pose such problems. (The direct translation from story problems to formal equations is a very difficult problem, and we think pedagogically less useful in the present connection.)

Wallace Feurzeig had the idea for this kind of treatment of story problems. George Lukas developed and extended the approach and wrote the associated LOGO programs with editorial assistance from Feurzeig. Philip Faflick and Jeffrey Tiedeman contributed some ideas and wrote preliminary versions of some of the programs.

2. Different Ways of Writing Equations

2.1 Infix Expressions and Prefix Expressions

The simplest arithmetic expressions consist of single numbers such as 234 or 4.27. Somewhat more complex forms involve two numbers separated by an operation symbol +, -, *, or / (standing for sum, difference, product, or quotient). Examples are 3/4, 23+49, 23*0.45, 371-246, each using a different one of the four operations. One way to look at such forms is to consider the operation +, -, *, or / as acting on the two numbers which surround it. (These are called the *operands*.)

How, in this way of looking at things, is a repeated sum like 1+2+3 to be interpreted? Is the '2' in the middle acted on by the first or the second of the '+' operators or by both of them? The answer becomes clear when we think of how we actually perform such a calculation. First we add 1 and 2 and then add the resulting number to 3. In other words, the operands of the first '+' are the numbers '1' and '2'. The two operands of the second '+' are 1+2 and 3. We thus see that arithmetic expressions can themselves be operands used by operators to form still more complex expressions. In the expression 3+5+7+9, the first (to the left) '+' operates on 3 and 5; the second on the result of that, 3+5, and 7; the operands of the third '+' are 3+5+7 and 9. In these repeated additions, we just go from left to right.

This, however, is not true for 2+3*4. Given this expression to calculate, we do not add 2 and 3, then multiply the result by 4. We could do it this way, but we ordinarily do it a different way - we perform the multiplication first, then add 2 to the result. Each operator, + and *, still has two operands (* acts on 3 and 4,

+ acts on 2 and 3*4). The order, however, is not simply left to right. If we wished to perform the operations in left to right order, we could so indicate by using parentheses as follows: (2+3)*4. But when there are no parentheses we adhere to the standard, well-established convention that multiplications are *performed* before additions even when a + is written to the left of some *. If we think of a few further cases, like 2+4/3+7*4 and 2-3*5+7-9 and 2-3+5*5, a pattern emerges. We first do all the multiplications and divisions. After this has been done we go from left to right, performing additions and subtractions, just as before. What about the case in which we have repeated multiplications and divisions? (2*4/5/6, for example.) Again, in such cases we just go from left to right, multiplying 2 and 4, dividing the result by 5, and dividing this new result by 6. Our rule is then: operations are performed from left to right, except that multiplications and divisions take precedence over additions and subtractions, regardless of order.

The familiar notation just described is, of course, not the only one used to write arithmetic expressions. In LOGO, for example, to perform the addition 3+5, we write "SUM OF 3 AND 5". Subtractions are written in the same way, e.g., "DIFFERENCE OF 7 AND 9". In both of these cases, the operation is written first, followed by the two operands. Any such notation, where the operation is written before the objects that it acts on is called *prefix** notation, the word 'prefix' referring to the position of the operation. The form in which the operation is placed between the objects is called *infix* notation.

---

*Sometimes called Polish notation after its inventor Lukasiewicz.

Infix notation is the standard one used in classroom and laboratory for writing arithmetic expressions, formulas, and equations. Prefix notation is used in many areas of mathematical work, for example in expressing functions and operators in analysis and logic. As well as infix and prefix notations, mathematicians and logicians sometimes employ *postfix* notation where the operands are written down first, followed by the operation. Thus 2+3, +2,3, and 2,3+ are infix, prefix, and postfix forms for the same expression.

In this discussion we shall be concerned with infix notation as well as two equivalent forms of prefix notation -- the one used in LOGO, and a compact one used to simplify the interpretation of complex expressions. In this compact prefix notation we write one of the operational symbols +, -, *, / followed by parentheses enclosing the two operands. Examples of simple expressions in this notation are +(2,3), -(7,9), *(5,6), /(2,17). Just as in the case of infix notation, we can create more complicated expressions by using arithmetic expressions themselves as operands.

We can, for example, use the first expression in the above example as an operand to write +(5,+(2,3)). We can use the second and third to write *(-(7,9),*(5,6)). (Upon evaluation, the first of these expressions gives 10, and the second -60.) This combining process, clearly, can be continued to create expressions of considerable complexity. Consider the relatively complicated expression, /((+(/(2,3),4)),*(5,6)). What does it mean? That is to say, how can one go about evaluating such an involuted prefix expression? We want to find a general procedure for evaluating any prefix expression. We'll try proceeding along the same lines that worked for evaluating arbitrarily complicated infix expressions. We'll first search for the simple subexpressions (i.e., for parts of this complicated expression whose operations can be performed right away without the knowledge of the values of other parts). There are two such subexpressions in this case /(2,3) and *(5,6). Once these are evaluated, the subexpressions containing these as operands can also be evaluated. Thus, the result of /(2,3) is used in the evaluation of +(/(2,3),4), and after this the final division can be performed.

2.2 Tree Diagrams

We can represent this evaluation process in a relatively transparent way using the following graphical scheme. To represent an operation on two numbers, we write the operation symbol, then the two operands below it, and connect each of them to the operation by straight lines. Figure 1a shows the representation of +(3,4) and Figure 1b that of /(5,6). We indicate the use of any prefix expression itself as an operand simply by repeating this process. Figure 2a, for example, represents *(+(3,4),5) and 2b, +(+(3,4),/(5,6)). Figure 3 shows the diagram corresponding to the more complicated expression we considered before /((+(/(2,3),4)),*(5,6)). These diagrams are called *tree diagrams*. Unlike oak trees, they broaden in a downward direction.

One way in which such diagrams are prepared takes the same steps as the evaluation procedure we just described. One first takes those pieces of the expression which can be evaluated immediately, and uses these in building up to the complete expression. In terms of tree diagrams this is called bottom-up writing, since we start at the bottom of the tree and work up. This is what we did in building up the final expression shown in Fig. 2b from Figs. 1a and 1b. Unlike evaluation, however, we can also create a tree from the top down. Let us reconsider the example depicted in Fig. 3 from this new point of view. The division operation furthest to the left in the expression is the last

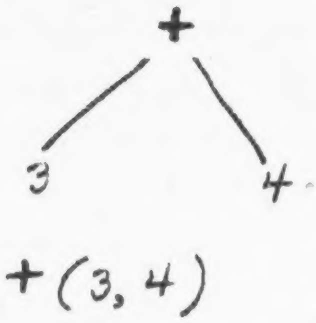


Fig. 1a.

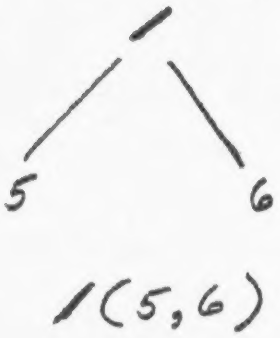


Fig. 1b.

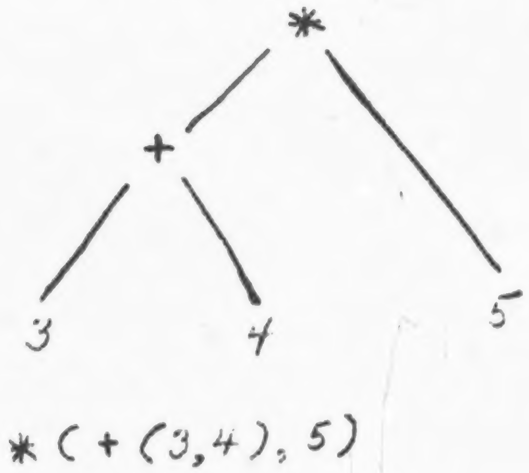


Fig. 2a.

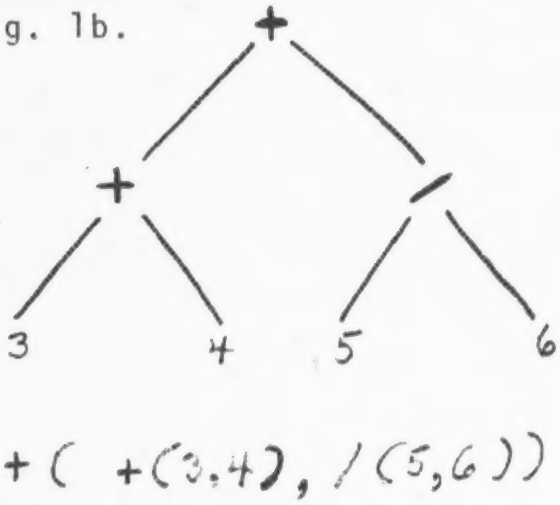


Fig. 2b.

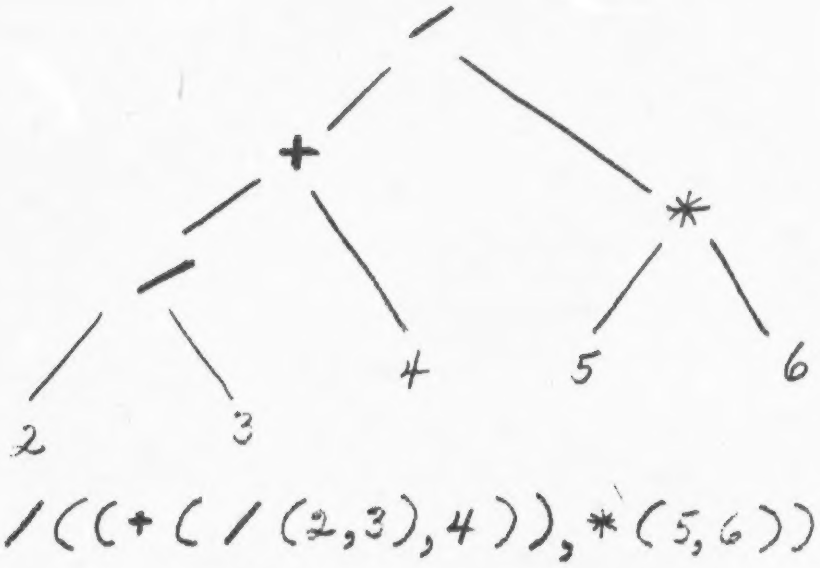

/
+
*
/
4
5
6
2
3
/((+(/(2,3),4)),*(5,6))

Fig. 3.

operation to be performed. This will certainly be the highest point on the tree so let us start with it. We see that this '/' operation involves a '+' and a '*', immediately preceding, so these are the next level down on the tree (see Fig. 4b). Continuing *down* , the '+' involves a '/' and the number 4. The '*' has as operands the numbers 5 and 6. The '/' has 2 and 3. These are included in Fig. 4c which shows the complete tree. The tree we get in this way is exactly the same as in Fig. 3. However, it was generated in the opposite way. This method of drawing a tree is called top-down.

Fig. 4a.

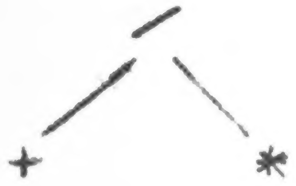


Fig. 4b.

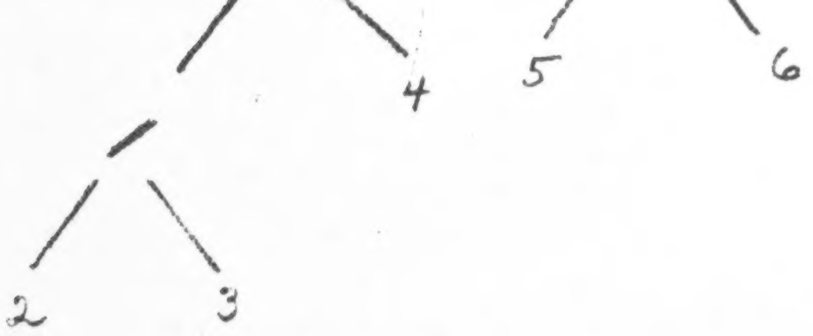


Fig. 4c.

When does a tree diagram represent a valid prefix expression? We can make some fairly straightforward observations about the necessary conditions for this. First, all operations in the tree must have two connecting downward lines, since each +, -, *, or / requires two objects for operands. Thus, the diagram in Fig. 5 clearly does not represent a prefix expression.

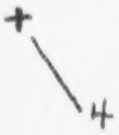


Fig. 5.

Further, since the innermost subexpressions of a prefix expression are simple operations on numbers, the last objects reached on any paths down from the top of the tree must be numbers (not operations). Thus, the situation in Fig. 6 is meaningless.

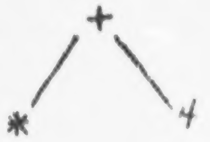


Fig. 6.

In fact, numbers must always be last objects and all the other objects in the *tree* must be operations. So the situation in Fig. 7 is meaningless too. (How do you perform the operation '4' on the numbers '5' and '6'?)

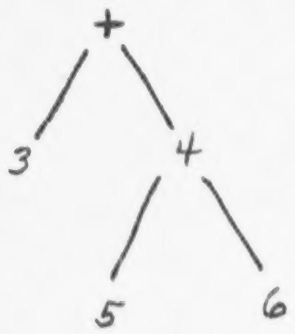


Fig. 7.

Last of all, the tree must have a unique starting point -- called its *root* operation. Thus, the diagram in Fig. 8 represents *two* expressions not a single one, though it can be made into a single one by connecting the roots of the two expressions together with a new operation (say -) as in Fig. 9. The two original expressions have become subexpressions in this extended diagram.

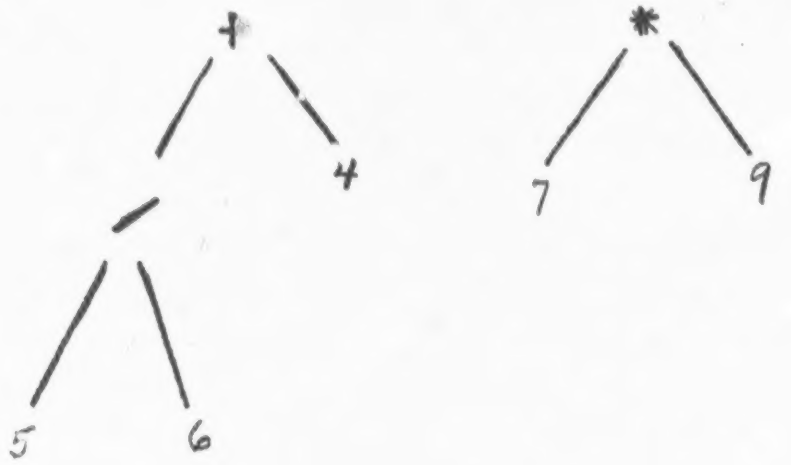


Fig. 8.

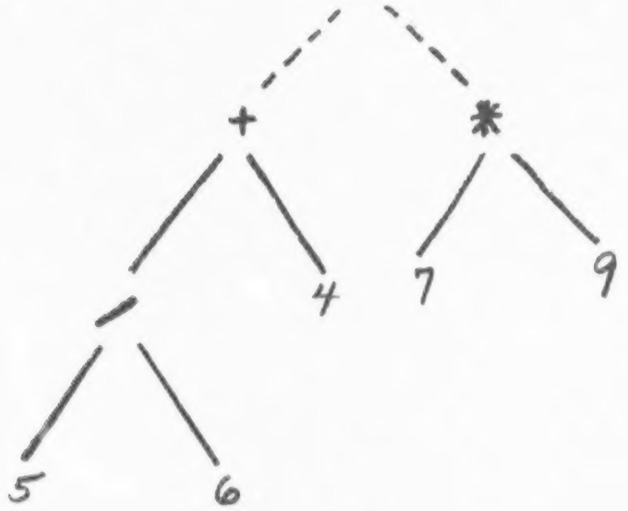


Fig. 9.

2.3 A Translation Problem

We now have two notations for writing arithmetic expressions. Superficially they look quite different. We might ask: are the two notations equally general, or are there some arithmetic expressions that can be written in one notation which we cannot express in the other notation? We can easily answer a related question comparing prefix notation with infix notation when we exclude the use of parentheses. We can immediately give examples to show that, *without parentheses*, infix notation is *less* general than prefix notation. A simple example of a prefix expression which has no equivalent form in "parenthesis-less" infix is *(+(2,3), 4). The corresponding tree diagram is drawn in Fig. 10.

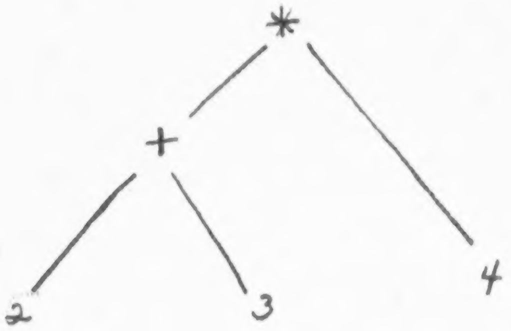


Fig. 10.

In words, we add 2 and 3, then multiply the result by 4. If we write 2+3*4 or 4*2+3 in infix, the multiplication is always done first. There is no way of indicating in infix notation that the addition is to be performed first unless we introduce parentheses. Using parentheses, we simply write (2+3)*4. (We also invoke the standard rule that operations within parentheses are always performed first.) The use of parentheses enables us to write expressions in which multiplications and divisions are performed before additions and subtractions. In fact, the use of parentheses gives us complete freedom to modify the order in which operations are performed in any arithmetic computation.

If we include parentheses, is infix now *more* general than prefix? The most concrete way to settle this question is to write a translation procedure for converting infix expressions into prefix form. If this translator works for all infix inputs, then we can say that infix is certainly *not* more general than prefix. The translator will give us a useful byproduct: if we translate infix expressions to the particular prefix form which LOGO uses (as distinct from other prefix forms), we can have LOGO evaluate infix expressions as well as prefix ones. Thus, we will be able to use LOGO to perform arithmetic calculations expressed in whichever notation is more appropriate to the problem at hand.

We shall start with a very simple case -- the translation of infix expressions of the form

operand1 + operand2

into the equivalent prefix form

SUM OF operand1 AND operand2.

Tentatively, we'll start with an approach requiring that the infix expression is written in a standard format. Let's assume that the infix expression our procedure is supposed to translate is written as a three-word LOGO sentence where the first word is one operand, the second word is "+", and the third and last word is the other operand. Given an input in this form, the procedure should output the four-part LOGO sentence:

"SUM OF" FIRST OF /INPUT/ "AND" LAST OF /INPUT/

The following one-line procedure, called PREFIX, incorporates this idea.

```
TO PREFIX /EXP/            (EXP reminds us of "expression")
10 OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF "SUM OF" (FIRST OF
 /EXP/) "AND" (LAST OF /EXP/)   (The parentheses are used merely
END                              to set off the parts of the
                                 expression.  They could have
                                 been omitted.)
```

Two examples illustrating the effect of PREFIX are:

```
←PRINT PREFIX "2 + 31"
SUM OF 2 AND 31

←PRINT PREFIX "SKUNK + CABBAGE"
SUM OF SKUNK AND CABBAGE
```

The inputs in both of these examples are sentences made up of three words -- but the procedure also gives the desired answer when the input is the single word "1+2", since FIRST and LAST, when applied to a word, output the first and last character of the word. Thus,

```
←PRINT PREFIX "1+2"
SUM OF 1 AND 2
```

As long as we keep to three-word sentences and three-character words, PREFIX works in the way we want it to. If this input restriction is violated, however, PREFIX gives funny outputs.

```
←PRINT PREFIX "23+45"
SUM OF 2 AND 5

←PRINT PREFIX "THREE ANTS + FOUR SPIDERS"
SUM OF THREE AND SPIDERS

←PRINT "2+ 2"
SUM OF 2+ AND 2    (FIRST OF "2+ 2" is 2+ and LAST OF "2+ 2" is 2)
```

The PREFIX procedure, as written, always takes the first part of any input expression as one operand and the last part as the other. Thus,

```
←PRINT PREFIX 56
SUM OF 5 AND 6

←PRINT PREFIX "2-3"
SUM OF 2 AND 3
```

We can sort out most of these questionable cases by modifying PREFIX as follows: First we add the instructions

```
>2 TEST IS COUNT OF /EXP/ 3
>4 IF FALSE OUTPUT "INPUT HAS WRONG NUMBER OF THINGS"
```

which check that the input contains exactly three words or characters. We can also have PREFIX check to make sure that the second (i.e., FIRST OF BUTFIRST) part of the input is the character + by adding the steps:

```
>6 TEST IS FIRST OF BUTFIRST OF /EXP/ "+"
>8 IF FALSE OUTPUT "THIS IS NOT AN INFIX ADDITION"
```

With these tests, our PREFIX procedure only reaches line 10, the translation step, for inputs which it can translate correctly. Any other inputs will result in an output "INPUT HAS WRONG NUMBER OF THINGS" or "THIS IS NOT AN INFIX ADDITION".

A natural question arises next: how can we generalize this procedure? We would like to be able to translate expressions whose operands are not single characters and also those which are not single words -- "23+45" and "THREE SPIDERS + FOUR MUSHROOMS" for example. This is not easy using our present approach, since it assumes that the two operands have fixed locations in the input -- for inputs which contain several characters or several words, the location of the operands will vary. In "234567+9", the first through sixth characters form the first operand and the eighth character the second operand. In "ONE + FOUR HIDEOUS GROTESQUE GROSSLY MISSHAPEN PENCILS", the first operand is just the first word, the second operand is the third through eighth words. Let us therefore look for other ways in which we can generalize PREFIX to deal with inputs of varying lengths.

We wish to modify the procedure so it can handle more complicated operands. Also, we'll try to get it to translate more complicated *kinds* of infix expressions. To begin with, let's extend PREFIX to include *repeated* summations such as "2+3+4+5", not just simple sums. An approach which builds on the existing PREFIX procedure is to go over a repeated sum several times, each time using PREFIX to translate one of the additions. On the first time around it would produce SUM OF 2 AND 3+4+5, on the second round, SUM OF 2 AND (SUM OF 3 AND 4+5) and finally, on the third round, it would output SUM OF 2 AND SUM OF 3 AND (SUM OF 4 AND 5) as the answer.

How can the present procedure be modified to perform these partial calculations? If we type,

```
PRINT PREFIX "2+3+4+5"
```

our present procedure responds:

```
INPUT HAS WRONG NUMBER OF THINGS
```

(meaning the input is either a word not containing three characters or a sentence not containing three words).

We could trick the procedure by writing

```
PRINT PREFIX  "2 + 3+4+5"
```

and get the desired result

```
SUM OF 2 AND 3+4+5
```

but the next time around the new input "3+4+5" would present the same problem we had with "2+3+4+5". This time there is no way around the difficulty -- we cannot use our trick twice. The PREFIX procedure would not have accepted 2 + 3 +4+5 as a valid input the first time around. Our original procedure does not seem to be easily generalized. The fault again seems to lie in depending upon recognizing operands by the exact position they occupy in the input. Let us, therefore, reconsider our whole approach to the translation problem.

We, ourselves, find it easy to translate an expression like 234534+242354 into prefix form. It is worthwhile, therefore, to ask how we do it. Given 234534+242354, we first look for the + sign and, finding it, write "sum of" (the part of the expression to the left of the +) "and" (the part to the right of the +), in this case obtaining "sum of 234534 and 242354". So the problem seems very naturally to separate into two simpler subproblems: finding everything to the left of the +, and finding everything to the right of the +. Let's try to use this approach in writing a LOGO procedure. We first write a procedure LEFT+ which outputs that portion of an input expression which is to the left of the + and then a procedure RIGHT+ which outputs the part of the input to the right of the +. Then all we need is a third procedure which combines the outputs of LEFT+ and RIGHT+. This last procedure can be simply written as:

```
TO PREFIX2 /EXP/
10 OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF "SUM OF" (LEFT+
  OF /EXP/) "AND" (RIGHT+ OF /EXP/)
END
```

An easy way to write LEFT+ is to have it examine the first character of the input. If it is not "+", then the output is the LOGO word made by joining this first character to LEFT+ of the rest of the word (i.e., its BUTFIRST). If, however, the first character *is* "+", then we output the EMPTY word as the answer, since there are no further characters before the "+". Using this idea we have:

```
TO LEFT+ /EXP/
10 TEST IS FIRST OF /EXP/ "+"
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT WORD OF (FIRST OF /EXP/) AND (LEFT+ OF BUTFIRST OF /EXP/)
END
```

Let's TRACE this procedure with /EXP/ taken to be "23+45".

```
←TRACE LEFT+ OF "23+45"
LEFT+ OF "23+45"
  LEFT+ OF "3+45"
    LEFT+ OF "+45"
    LEFT+ OUTPUTS ""
  LEFT+ OUTPUTS "3"
LEFT+ OUTPUTS "23"
←
```

What happens when there is more than one + in the input expression?

```
←PRINT LEFT+ "2+3+4"
2
```

LEFT+ tests whether the leftmost character of the input is "+"; if not, it tests the next character and keeps doing this till it finds a +. When it reaches the first + it stops searching. Thus, the output is everything to the left of the first +. This is true even if there are other operations in the input:[1]

```
←PRINT LEFT+ "2*3/4+5"
2*3/4
```

Let us try LEFT+ on a sentence, instead of a word

```
PRINT LEFT+ "ONE APPLE + TWO PEARS + THREE APPLES"
ONEAPPLE
```

LEFT+ ran together the two words to the left of the first +. We did not see this behavior in our previous examples using LEFT+ since all previous inputs were LOGO *words*. LEFT+ examines

---

[1] By the same process:

```
←PRINT LEFT+ "(1+2)+3"
(1
```

and parentheses obviously could cause serious difficulties in translation. Parentheses really become important only when more than one kind of operation occurs in an expression. We will delay our treatment of parentheses until we have extended our translation to include all four standard arithmetic operations.

FIRST OF /EXP/ to see if it is a +. If /EXP/ is a *sentence*, then FIRST OF /EXP/ is a *word*, if /EXP/ is a *word*, then FIRST OF /EXP/ is a character. In either case, LEFT+ outputs the *word* created by combining the first element of /EXP/ with LEFT+ of the rest of /EXP/. We want LEFT+ to output WORD OF FIRST /EXP/ AND LEFT+ BUTFIRST /EXP/ only if /EXP/ itself is a word. If /EXP/ is a sentence, we should output SENTENCE OF FIRST /EXP/ AND LEFT+ BUTFIRST /EXP/. This is easily done by testing whether /EXP/ is a word -- TEST WORDP /EXP/ -- before outputting. (WORDP is a procedure which outputs "TRUE" if its input is a word and "FALSE" if its input is a sentence.) If /EXP/ is a word, then we want LEFT+ to output as before, using WORD to combine FIRST /EXP/ and LEFT+ of the rest of /EXP/. If /EXP/ is a sentence, we should use SENTENCE OF, rather than WORD OF, as the combining operation. Making this change, LEFT+ looks like:

```
TO LEFT+ /EXP/
10 TEST IS FIRST OF /EXP/ "+"
20 IF TRUE OUTPUT /EMPTY/
30 TEST WORDP /EXP/
40 IF TRUE OUTPUT WORD OF
    FIRST /EXP/
    LEFT+ BUTFIRST /EXP/
50 OUTPUT SENTENCE OF
    FIRST /EXP/
    LEFT+ BUTFIRST /EXP/
END
```

```
←PRINT LEFT+ "ONE APPLE + TWO PEARS + THREE APPLES"
ONE APPLE
```

Another interesting situation is when the input does not contain a + at all. Then the procedure LEFT+ behaves quite differently -- it seems to go on endlessly without giving an output. Let's trace LEFT+ and see how it works on inputs without +.

```
←TRACE LEFT+
←LEFT+ "234"
```

results in:

```
LEFT+ "234"
  LEFT+ "34"
    LEFT+ "4"
      LEFT+ ""
        LEFT+ ""
          LEFT+ ""
              .
              .
              .
```

and so on.

Now we can see what is happening. Once the expression becomes /EMPTY/, LEFT+ reaches an impasse and keeps making the same call fruitlessly. This is what is happening: LEFT+ keeps testing whether FIRST of the expression being considered is "+". Since this expression is BUTFIRST OF /EMPTY/ which itself is /EMPTY/, the condition is never satisfied and LEFT+ keeps going indefinitely. (Actually after a long time, the computer prints an error message and stops.)

Normally, LEFT+ outputs whatever is to the left of the first + encountered in its input. If the input does not contain a + at all, what *should* we output as the answer? We have to decide what to do in this special case, else the procedure will go on indefinitely as we just saw. If we determine that the input has no +, we *could* output an error message such as "THIS EXPRESSION HAS NO +". That seems a *reasonable* choice. But what if the expression was something like "SUM OF 2 AND 3" or "TWO APPLES" or just a number? We might be happier if the final translation by the procedure PREFIX (which we are in the process of writing) left such expressions unchanged -- even if they were not in standard infix form to begin with. This choice would give us the following results for the examples just given.

```
←PRINT PREFIX "SUM OF 2 AND 3"
SUM OF 2 AND 3
```

```
←PRINT PREFIX "TWO APPLES
TWO APPLES

←PRINT PREFIX "32"
32
```

Thus we will adopt the choice that LEFT+ will output the entire input if the input does not contain a +. Considering our desired goal, this seems to be a reasonable choice.

To accomplish the result in LEFT+, we need only test whether the input has become /EMPTY/. If it has, we do not want LEFT+ to continue looking for a +. So we have it output /EMPTY/, as the answer for the *current* input. The modified procedure LEFT+ is now as follows.

```
TO LEFT+ /EXP/
3 TEST IS /EXP/ /EMPTY/       (If the input is /EMPTY/, the answer
6 IF TRUE OUTPUT /EMPTY/       is /EMPTY/)
10 TEST IS FIRST /EXP/ "+"    (Otherwise, the procedure is as before)
20 IF TRUE OUTPUT /EMPTY/
30 TEST WORDP /EXP/
40 IF TRUE OUTPUT WORD OF
     FIRST /EXP/
     LEFT+ BUTFIRST /EXP/
50 OUTPUT SENTENCE OF
     FIRST /EXP/
     LEFT+ BUTFIRST /EXP/
END
```

Now let's see what happens when we trace LEFT+ "234".

```
LEFT+ OF "234"
  LEFT+ OF "34"
    LEFT+ OF "4"
      LEFT+ OF ""
      LEFT+ OUTPUTS ""
    LEFT+ OUTPUTS "4"
  LEFT+ OUTPUTS "34"
LEFT+ OUTPUTS "234"
```

This time the process terminates and produces the desired output.

We still have a small problem with some inputs, however. The procedure LEFT+ keeps on testing whether FIRST of the input is "+". This means that if the input is a *word* (in the LOGO sense), LEFT+ will always find a + in the input if there is one. If, however, the input is a *sentence*, each *word* is teated to see if it is "+". So if the + is not a separate whole word, it is not found. For instance,

```
←TRACE LEFT+
←LEFT+ "2+ 3"          ("2+ 3" is a two-word sentence)
```

results in:

```
LEFT+ OF "2+ 3"       (FIRST of "2+ 3" is "2+", which is not "+",
  LEFT+ OF "3"         and BUTFIRST of "2+ 3" is the second word,
    LEFT+ OF ""        "3")
    LEFT+ OUTPUTS ""
  LEFT+ OUTPUTS "3"
LEFT+ OUTPUTS "2+ 3"     (Instead of 2, as desired)
```

To remedy this, we could write a procedure PACK which, given a sentence as input, squeezes together all its words using the operation WORD. In the process, it eliminates all spaces, and changes the LOGO sentence to a LOGO word.

```
TO PACK /EXP/
10 TEST IS /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT WORD OF (FIRST OF /EXP/) AND (PACK OF BUTFIRST OF /EXP/)
END

←PRINT PACK OF "2+ 3"
2+3
```

We could include PACK as part of PREFIX by inserting in PREFIX a step like:

```
1 MAKE
    NAME:  "EXP"
    THING:  PACK OF /EXP/
```

Then LEFT+ would always work correctly.

The use of PACK in PREFIX would eliminate most format problems. Like most automatic standardization procedures, however, it would give us less direct control over the situation it helps us with. For example, the use of PACK would squash together any sentences used as inputs.

```
←PREFIX "TWO APPLES + THREE PEARS"
SUM OF TWOAPPLES AND THREEPEARS
```

We can either content ourselves with this state of affairs and not use sentence inputs which we don't want 'squashed', or we can *omit* PACK *from* PREFIX and take special care in how we write our input expressions -- if our input is a sentence, we must make sure that all + signs are written as separate words, surrounded by spaces, so they will not be passed over and ignored.

If we *really* want to concern ourselves with expressions like "11+ 3 RED HERRINGS +35" we can write a procedure SEPARATE+, which puts spaces on both sides of any + which does not already have them, for example,

```
←SEPARATE+ "11+ 3 RED HERRINGS +35"
11 + 3 RED HERRINGS + 35
```

This procedure is included in the appendix.

Only RIGHT+ remains to be written to complete PREFIX. It is written along much the same lines as LEFT+. Here again, we test the first character of the input, checking whether it is "+". If not, we repeat, checking BUTFIRST of the input. When a "+" is found, we want to output everything after it -- that is, BUTFIRST of the input.

```
TO RIGHT+ /EXP/
10 TEST IS FIRST OF /EXP/ "+"
20 IF TRUE OUTPUT BUTFIRST OF /EXP/
30 OUTPUT RIGHT+ OF BUTFIRST OF /EXP/
END
```

Note that unlike LEFT+, word and sentence inputs are both treated correctly, without any distinction having to be made between them, since no combining operation is needed to produce the correct output.

We do, however, have a problem with this version of RIGHT+ like one that came up with the corresponding version of LEFT+ -- that if the input contains no +, the procedure just keeps on going. Since, in such cases, LEFT+ already outputs the entire input, it is reasonable for the procedure RIGHT+ to output the empty word. To do this, we simply check whether the input is empty and, if so, output /EMPTY/. The final version of RIGHT+ is therefore:

```
TO RIGHT+ /EXP/
10 TEST IS /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 TEST IS FIRST OF /EXP/ "+"
40 IF TRUE OUTPUT BUTFIRST OF /EXP/
50 OUTPUT RIGHT+ OF BUTFIRST OF /EXP/
END
```

All its parts are completed so PREFIX is ready to be used. Given any expression containing a single +, it translates the addition into prefix notation and outputs the result. Thus,

```
←PRINT PREFIX "2345+6789"
SUM OF 2345 AND 6789

←PRINT PREFIX "THREE APPLES + ONE PEAR"
SUM OF THREE APPLES AND ONE PEAR
```

Let's now try PREFIX with an input which does not have a +.

```
←PRINT PREFIX "23456"
SUM OF 23456 AND
```

The explanation for this strange printout is as follows: LEFT+ "23456" is 23456, the entire word, and RIGHT+ "23456" is the empty word. In the case of such inputs, containing no +, we want PREFIX to print the input itself as the result. We can easily change PREFIX appropriately by testing whether LEFT+ OF /EXP/ is the same as /EXP/.* If this is true, it means there is no + in the input expression and we can output the input expression unchanged. Otherwise, the output is as before.

```
TO PREFIX /EXP/
10 TEST IS (LEFT+ OF /EXP/) /EXP/
20 IF TRUE OUTPUT /EXP/
30 OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF "SUM OF" (LEFT+
   OF /EXP/) "AND" (RIGHT+ OF /EXP/)
END
```

```
←PRINT PREFIX "A PARTRIDGE IN A PEAR TREE"
A PARTRIDGE IN A PEAR TREE
```

We have generalized PREFIX from its original form to enable us to handle objects of arbitrary length. Let's consider the other generalization we mentioned earlier -- translating more complicated infix forms such as repeated additions. PREFIX accepts forms of this kind and does *something* with them. Thus,

```
←PRINT PREFIX "3+5+7+9"
SUM OF 3 AND 5+7+9
```

This is a good beginning. Let's apply PREFIX to the remaining untranslated portion.

---

*Perhaps a more straightforward way to determine whether /EXP/ has a + is to write a new procedure EXIST+ /EXP/ which searches /EXP/ for a + and outputs "TRUE" or "FALSE" depending on whether or not /EXP/ contains a +.

```
←PRINT PREFIX "5+7+9"
SUM OF 5 AND 7+9
```

Again applying PREFIX to the last untranslated part -

```
←PRINT PREFIX "7+9"
SUM OF 7 AND 9
```

We see that if we keep applying PREFIX to the untranslated portion of a multiple sum, eventually the entire expression gets translated. We can get PREFIX to do the complete translation by itself through a remarkably simple recursion. All we need do is replace RIGHT+ OF /EXP/ on line 40 with PREFIX OF RIGHT+ /EXP/. Instead of simply outputting whatever expression is on the right of the *first* +, as before, the procedure goes on to translate this expression too. The translation continues in this way until there are no more +'s remaining. With this change, PREFIX becomes:

```
TO PREFIX /EXP/
10 TEST IS (LEFT+ OF /EXP/) /EXP/
20 IF TRUE OUTPUT /EXP/
30 OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF "SUM OF"
   (LEFT+ OF /EXP/) "AND" (PREFIX OF RIGHT+ OF /EXP/)
END
```

To see how this extended version of PREFIX works, let's trace it:

```
←TRACE PREFIX
←PREFIX "3+5+7+9"
PREFIX OF "3+5+7+9"
  PREFIX OF "5+7+9"
    PREFIX OF "7+9"
      PREFIX OF "9"
      PREFIX OUTPUTS "9"
    PREFIX OUTPUTS "SUM OF 7 AND 9"
  PREFIX OUTPUTS "SUM OF 5 AND SUM OF 7 AND 9"
PREFIX OUTPUTS "SUM OF 3 AND SUM OF 5 AND SUM OF 7 AND 9"
```

So, just by adding a single word, we have extended our procedure to translate repeated summations of any length.

Consider what PREFIX does to expressions containing a "-".

```
←PRINT PREFIX "3+4-2+3"
SUM OF 3 AND SUM OF 4-2 AND 3
```

This is an excellent result. PREFIX correctly translates all the + parts and leaves the - part in infix form. Thus, this version of PREFIX can be a useful part of a more general procedure for translating any prefix expression containing both +'s and -'s.

2.4 Extending the Infix-Prefix Translator

We want to develop a procedure which can translate any infix expression into a corresponding prefix form. As a first step in the extension of the PREFIX procedure we have already written, let us generalize it to handle differences as well as sums. A reasonable way to do this is by writing a set of procedures which can translate repeated differences, just as the present version of PREFIX translates repeated sums, and then combining the two sets of procedures to form a general sum-difference translator.

Our strategy in writing such a repeated difference translator can be exactly the same as we used with repeated sums. We want to translate expressions of the form *operand1 - operand2* into DIFFERENCE OF *operand1* AND *operand2*. We again require two procedures, one which we call LEFT- which, given an arbitrary expression as input, outputs that portion of the input expression to the left of the first "-" sign, and a procedure RIGHT- which outputs that portion of the input expression to the right of the

142

first "-". Rather than write two such procedures from the beginning we note that they are just the same as the corresponding procedures RIGHT+ and LEFT+ that we wrote for additions, except that here we have a "-" where previously we had a "+". Let us therefore write two *general* procedures LEFT /SYMBOL/ /EXP/ and RIGHT /SYMBOL/ /EXP/, which give as output that portion of /EXP/ to the left and right of the first occurrence of /SYMBOL/, where /SYMBOL/ can be +, -, or any other character. These can be used in place of LEFT+, RIGHT+, LEFT-, and RIGHT- and will probably be useful later on, as well, when we are working to extend our translation capabilities to include multiplication and division operations. The changes we need to make in LEFT+ and RIGHT+ to generalize them to LEFT /SYMBOL/ /EXP/ and RIGHT /SYMBOL/ /EXP/ are quite minor. The title lines must be changed, of course, to take into account the change in procedure name as well as the increase in the number of inputs. In line 10 in each procedure, where we test whether the first character of /EXP/ is +, we now want to check whether the first character of /EXP/ is /SYMBOL/. Also, let us not forget to change lines 40 and 50 to take into account the new procedure name. We have, therefore,

```
TO LEFT /SYMBOL/ /EXP/                     (Title line is changed)
3 TEST IS /EXP/ /EMPTY/
6 IF TRUE OUTPUT /EMPTY/
10 TEST IS FIRST /EXP/ /SYMBOL/            (Second change)
20 IF TRUE OUTPUT /EMPTY/
30 TEST WORDP /EXP/
40 IF TRUE OUTPUT WORD OF
     FIRST OF /EXP/ AND
     LEFT OF /SYMBOL/ AND (BUTFIRST OF /EXP/)  (The new procedure
50 OUTPUT SENTENCE OF                          name LEFT is used
     FIRST OF /EXP/                            here)
     LEFT OF /SYMBOL/ AND (BUTFIRST OF /EXP/)
END
```

```
TO RIGHT /SYMBOL/ /EXP/
3 TEST IS /EXP/ /EMPTY/
6 IF TRUE OUTPUT /EMPTY/
10 TEST IS FIRST /EXP/ /SYMBOL/
20 IF TRUE OUTPUT BUTFIRST OF /EXP/
30 OUTPUT RIGHT /SYMBOL/ BUTFIRST /EXP/
END
```

These procedures are easily checked:

```
←PRINT LEFT "X" "FOXNOXSOX"
FO
←PRINT RIGHT "X" "FOXNOXSOX"
NOXSOX
←PRINT LEFT "*" "A+B+C"
A+B+C
←PRINT RIGHT "*" "A+B+C"
←
```

and they seem to be correct.

We can now write the general procedure PREFIX-, to translate repeated differences from infix to prefix. Again, we pattern this after the corresponding PREFIX procedure used for additions. We have to change the procedure name references -- we now have LEFT and RIGHT, each with two inputs, instead of LEFT+ and RIGHT+ with one, and the procedure itself is named PREFIX- instead of PREFIX. Also, everywhere we had + or SUM we now want - or DIFFERENCE. With these changes we have:

```
TO PREFIX- /EXP/
10 TEST IS
    LEFT "-" /EXP/
    /EXP/
20 IF TRUE OUTPUT /EXP/
30 OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
    "DIFFERENCE OF"
    PREFIX- LEFT "-" /EXP/
    "AND"
    PREFIX- RIGHT "-" /EXP/
END
```

We again test our result:

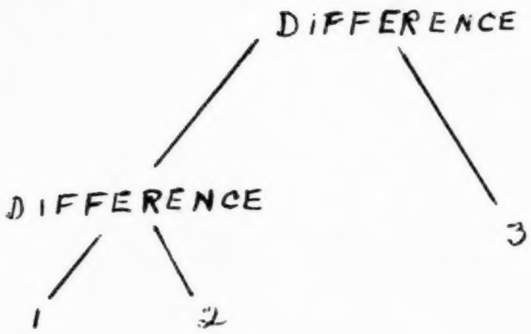
```
←PRINT PREFIX- "1-2-3"
DIFFERENCE OF 1 AND DIFFERENCE OF 2 AND 3
←
```

This is not right. We can see this either by translating the given infix "1-2-3" mentally or by noting that the infix expression when evaluated gives -4 and the output of PREFIX- gives 2. (The difference of 2 and 3 is -1, the difference of 1 and -1 is 2.) The correct prefix expression for the given infix input is DIFFERENCE OF DIFFERENCE OF 1 AND 2 AND 3. We can get some idea of how PREFIX- arrives instead at an unacceptable output by using a TRACE. First, for simplicity, let us just TRACE PREFIX- as it operates on the input "1-2-3", used above. Then, if more information seems necessary, we can also trace LEFT and RIGHT.

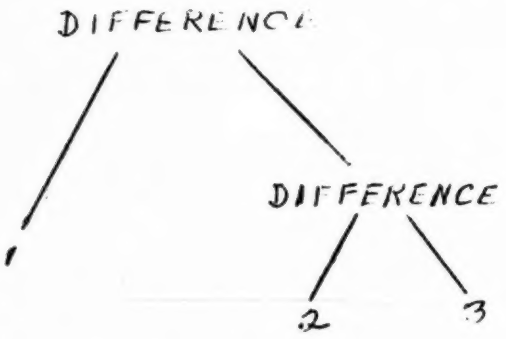
```
TRACE PREFIX-
PREFIX- "1-2-3"
PREFIX- OF "1-2-3"
  PREFIX- OF "2-3"
    PREFIX- OF "3"
    PREFIX- OUTPUTS "3"
  PREFIX- OUTPUTS "DIFFERENCE OF 2 AND 3"
PREFIX- OUTPUTS "DIFFERENCE OF 1 AND DIFFERENCE OF 2 AND 3"
```

(We needn't use PRINT since all outputs of PREFIX- are PRINTed in the course of TRACing)

The rule for calculating the result of repeated subtractions is that one goes from left to right. In the case being considered, we should start with the 1, subtract 2 from it -- DIFFERENCE OF 1 AND 2, then subtract 3 from this -- DIFFERENCE OF (DIFFERENCE OF 1 AND 2) AND 3. The procedure we have just written goes in just the opposite direction -- it performs the righthand subtraction first. In terms of tree diagrams we have:



what we should get



what we got

This seems a little strange: after all, PREFIX- looks first for the leftmost - sign (since LEFT ("-" /EXP/) and RIGHT ("-" /EXP/) go from left to right in searching for a -), and, having found one, looks for the next - sign to the right, etc. Why, then, does the output give the result of taking subtractions from right to left? The answer can be seen from the trace. It is true that PREFIX- goes from left to right in *looking* for - signs. The *translating*, however, only starts when the rightmost - has been reached -- the last PREFIX- called is the first to OUTPUT. The result is that the expression is looked at (in fancy terms, 'parsed') from left to right, which results, in our scheme of things, in a translation from right to left which is the opposite of what we want. Now that our difficulty is seen, the remedy should be apparent. If we just have PREFIX- look for - signs from right to left, then the translation will be in the correct order. This requires that RIGHT and LEFT be rewritten so that for inputs /SYMBOL/ /EXP/ they give the portion of /EXP/ to the right and left of the *rightmost* occurrence of /SYMBOL/. In other words, we want a RIGHT and LEFT such that RIGHT OF "X" AND "FOXNOXSOX" results in /EMPTY/ as output and LEFT "X" "FOXNOXSOX" outputs FOXNOXSO. What modifications are required in the existing versions? In our new version of LEFT, we want to compare LAST /EXP/ with /SYMBOL/. If they match, LEFT should output BUTLAST /EXP/ -- that is, everything to the left of /SYMBOL/.

If the last character of /EXP/ is not /SYMBOL/, then we should repeat the procedure with BUTLAST /EXP/ in place of /EXP/. Our new procedure LEFT is, then,

```
TO LEFT /SYMBOL/ /EXP/
3 TEST IS /EXP/ /EMPTY/
6 IF TRUE OUTPUT /EMPTY/
10 TEST IS LAST /EXP/ /SYMBOL/
20 IF TRUE OUTPUT BUTLAST /EXP/
30 OUTPUT LEFT /SYMBOL/ BUTLAST /EXP/
END
```

The changes in RIGHT are similar to the ones made in LEFT. We again look at the last character of /EXP/. If it is /SYMBOL/, then there is nothing to the right of /SYMBOL/ and we output /EMPTY/. If the last character of /EXP/ is not /SYMBOL/, then RIGHT should output the word composed of RIGHT applied to BUTLAST /EXP/ and the last character of /EXP/.

```
TO RIGHT /SYMBOL/ /EXP/
3 TEST IS /EXP/ /EMPTY/
6 IF TRUE OUTPUT /EMPTY/
10 TEST IS LAST /EXP/ /SYMBOL/
20 IF TRUE OUTPUT /EMPTY/
30 TEST WORDP /EXP/
40 IF TRUE OUTPUT WORD OF
    RIGHT /SYMBOL/ BUTLAST /EXP/
    LAST /EXP/
50 OUTPUT SENTENCE OF
    RIGHT /SYMBOL/ BUTLAST /EXP/
    LAST /EXP/
END
```

We need not change anything in PREFIX- itself, so we are all done. Trying PREFIX- again, we now get:

```
←PRINT PREFIX- "1-2-3"
DIFFERENCE OF DIFFERENCE OF 1 AND 2 AND 3
←
```

We also want PREFIX-, as before, to output expressions in which no - occurs, unchanged.

```
←PRINT PREFIX- "ABCDE"
DIFFERENCE OF   AND ABCDE
←
```

The reason for this result, as we can see from looking back at PREFIX-, is that our test in line 10 for nonexistence of - in /EXP/ no longer seems to work. And indeed it does not.

```
←PRINT LEFT "-" "ABCDE"
←
```

LOGO responds by typing the empty word. So, if /SYMBOL/ does not appear in /EXP/, /EMPTY/ is output by LEFT.

```
←PRINT RIGHT "-" "ABCDE"
ABCDE
←
```

We could also have predicted this behavior just by thinking carefully about how each of these two procedures work. We can change this simply by changing the test for nonexistence of -, line 20 of PREFIX-, to

```
20 TEST IS
    RIGHT "-" /EXP/
    /EXP/
```

and now,

```
←PRINT PREFIX- "ABCDE"
ABCDE
←PRINT PREFIX- "1+2+3"
1+2+3
←
```

PREFIX- is seen to work for all the sorts of input of interest to us.

We have a version of PREFIX which correctly translates repeated infix sums to a corresponding prefix form. Also, we now have PREFIX- to do the same for repeated differences. Our next step, it appears, should be merging of these two procedures to enable us to translate automatically all expressions containing both + and - symbols.

We again want to look at the expression from the right, so that the translation, which starts with the last application of PREFIX+- is from left to right. Before, in our PREFIX and PREFIX- procedures, this was easy because RIGHT and LEFT take care of this as they stand. Now, however, we want to look at the symbol farthest to the right, and we cannot know, as we write the procedure, whether it will be + or -. The most straightforward way to solve this problem is to write a new procedure FINDR+- /EXP/ which, given input /EXP/ tells us which of the symbols + and - appears furthest to the right. If neither appear, a reasonable output is /EMPTY/. Using this FINDR+- and the RIGHT and LEFT which we have already written, PREFIX+- can be written as:

```
TO PREFIX+- /EXP/
10 TEST IS FINDR+- /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EXP/        (If neither + or - appear in /EXP/
30 TEST IS FINDR+- /EXP/ "+"     we output /EXP/ unchanged)
40 IF TRUE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
    "SUM OF"
    LEFT "+" /EXP/
    "AND"
    RIGHT "+" /EXP/
50 OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
    "DIFFERENCE OF"
    LEFT "-" /EXP/
    "AND"
    RIGHT "-" /EXP/
END
```

As written, PREFIX+- only works once -- it translates only the + or - operation furthest to the right. To correct this we do not want to output, in lines 40 and 50 just the LEFT and RIGHT of the appropriate symbol, but we want to apply PREFIX+- again to these parts of the given expression. Thus, underlining additions,

```
40 IF TRUE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
    "SUM OF"
    PREFIX+- LEFT "+" /EXP/
    "AND"
    PREFIX+- RIGHT "+" /EXP/
50 OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
    "DIFFERENCE OF"
    PREFIX+- LEFT "-" /EXP/
    "AND"
    PREFIX+- RIGHT "-" /EXP/
```

To finish this part of our translation problem we need only write the procedure FINDR+-. FINDR+- /EXP/ will output + or -, whichever is farther to the right in /EXP/. If neither appears, FINDR+- is to output /EMPTY/. To do this, we have FINDR+- look at the last character of /EXP/. If it is + or -, we are done and we output whichever one it is. If it is neither, we apply FINDR+- to BUTLAST /EXP/. If we get to the point where /EXP/ is /EMPTY/, it means we have gone through the initial expression without encountering a + or - and, therefore, if /EXP/ is /EMPTY/, we output /EMPTY/.

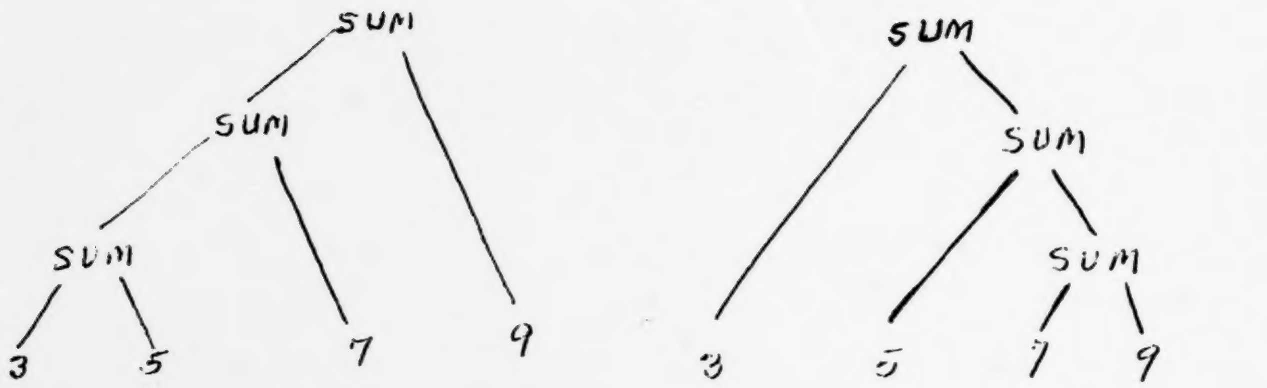
```
TO FINDR+- /EXP/
10 TEST IS /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 TEST EITHER
    IS LAST /EXP/ "+"
    IS LAST /EXP/ "-"
40 IF TRUE OUTPUT LAST /EXP/
50 OUTPUT FINDR+- BUTLAST /EXP/
END
```

and we are all done!

Now let us try the completed PREFIX+-.

```
←PRINT PREFIX+- "1-2-3+4*5"
SUM OF DIFFERENCE OF DIFFERENCE OF 1 AND 2 AND 3 AND 4*5

←PRINT PREFIX+- "123456789"
123456789
```

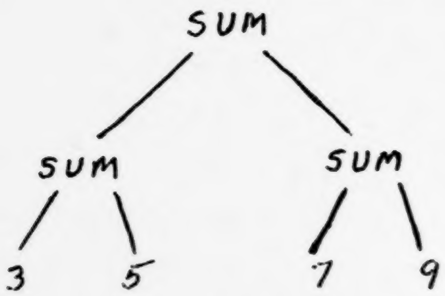
An interesting question to raise at this point is: why did the PREFIX procedure we wrote for sums appear to work, when we saw from dealing with differences that the translation proceeded in the wrong direction? Let us look back at an example: PREFIX "3+5+7+9" output SUM OF 3 AND SUM OF 5 AND SUM OF 7 AND 9. The tree for this is shown below. According to our rule that one does additions and subtractions going from left to right, this answer is not correct. The first thing to be done by that rule is SUM OF 3 AND 5, then 7 should be added to that and then 9 to this partial sum. Thus, the answer should really be SUM OF (SUM OF (SUM OF 3 AND 5) AND 7) AND 9. The tree corresponding to this is also drawn below. The reason this apparent mistake is hard to catch, and in fact is not really a mistake at all, is that it doesn't matter in which order additions are performed. The two prefix results given above give the same answer -- 24, and in fact, any way of adding the four quantities gives 24. For example, we might take: SUM OF (SUM OF 3 AND 5) AND (SUM OF 7 AND 9). This property of the + operator, of giving the same answer no matter which operands are added first, as long as order is preserved, is called associativity. An interesting way of putting this is that given the arguments, as in Fig. 11, any tree in which the operations are all sums gives the same value upon evaluation.



The tree which is equivalent to adding from left to right

The tree equivalent to the prefix we got using PREFIX

Both give an answer of 24 when the additions are performed.



Also gives the same result, 24.

Fig. 11.

Of course, as we already saw, subtraction does not possess this property of associativity and, given a string of differences, it certainly matters in which order they are performed. The fact that subtraction is not associative and that addition is can be used to considerably simplify the sum-difference translation problem. We see that we must perform all the subtractions strictly left to right but that the additions can be done in any order. Therefore, we can translate all the subtractions first, left to right, and the additions afterwards. This way, the associativity of subtraction is taken into account and we know we can do the additions in any order. We can use this associativity property of addition to write a considerably simpler version of PREFIX+- without requiring the use of FINDR+-. The simplification is accomplished by translating all the additions first, in whatever order they lie, and then the subtractions, right to left, as before:

```
TO PREFIX+- /EXP/
10 TEST IS (RIGHT "+" /EXP/) /EXP/      (Is there a '+' in /EXP/)
20 IF FALSE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
    "SUM OF"                            (If so, translate it and
    PREFIX+- LEFT "+" /EXP/             apply PREFIX+- to either
    "AND"                               side)
    PREFIX+- RIGHT "+" /EXP/
30 TEST IS (RIGHT "-" /EXP/) /EXP/      (Is there a - in /EXP/)
40 IF FALSE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
    "DIFFERENCE OF"                     (If so, translate it and
    PREFIX+- LEFT "-" /EXP/             apply PREFIX+- to either
    "AND"                               side)
    PREFIX+- RIGHT "-" /EXP/
50 OUTPUT /EXP/                         (If neither + nor - appears,
END                                     output /EXP/)
```

```
←PRINT PREFIX+- "ONE ELEPHANT + 2X - 143"
DIFFERENCE OF SUM OF ONE ELEPHANT AND 2X AND 143
```

There is only one note of caution in the use of the new PREFIX+- procedure. It assumes that the input is either a LOGO word or else a LOGO sentence each of whose operands appears as a LOGO word. We can guarantee the former assumption by using the procedure PACK discussed above, and the latter by writing a procedure SEPARATE to separate all operands as individual words. The SEPARATE procedure is shown in the appendix. To use it, we need merely add the line

```
5 MAKE "EXP" SEPARATE /EXP/
```

to PREFIX+-. With this modification, the input can be in any form.

2.5 Evaluating Infix Expressions Automatically

We can use PREFIX+- to translate any given series of sums and differences in infix notation to an equivalent prefix expression. The precise form of prefix notation that is given by PREFIX+-, furthermore, is exactly that used by LOGO. A natural extension of our capabilities in dealing with infix expressions, then, is to enable the computer to evaluate the prefix expressions output by PREFIX+-. This should not be too difficult since the expressions put out by PREFIX+- are already valid LOGO instructions. Let us consider, for example, the input "1+2-4" to PREFIX+-:

```
←PRINT PREFIX+- "1+2-4"
DIFFERENCE OF SUM OF 1 AND 2 AND 4
←
```

If we copy this output and put a PRINT command in front of it, it looks like an ordinary, perfectly valid LOGO instruction,

```
PRINT DIFFERENCE OF SUM OF 1 AND 2 AND 4
```

and it can be used as such. If this instruction is executed, LOGO will evaluate the expression and type the result:

```
-1
```

Doing the evaluation this way, we have to copy the output of PREFIX+- over again, and give LOGO this (possibly very lengthy) prefix expression as an input to a PRINT instruction. Instead, we would like LOGO to do this work for us automatically. We can accomplish this by using the LOGO command DO. DO is a command which has one input. DO treats its input as a LOGO instruction, i,e., its effect is to interpret and perform the input as an instruction line. Here are some simple examples to illustrate the operation of DO:

```
←PRINT SUM OF 2 AND 2
4
←DO "PRINT SUM OF 2 AND 2"
4
←DO SENTENCE OF "PRINT" AND "SUM OF 2 AND 2"
4
```

In the last two cases, DO interpreted its input as an instruction, not just a literal, and then performed it.

The operation of DO is more useful in the situation of interest to us. What we want to do is make a sentence of "PRINT" and the output of PREFIX+- and then perform this sentence as an instruction. So let's just do it:

```
←DO SENTENCE OF "PRINT" AND PREFIX+- "1+2-4"
-1
```

DO works as above in the simpler examples, only here it accomplishes something substantial for us on the way -- the evaluation of the output of PREFIX+- "1+2-4", that is of DIFFERENCE OF SUM OF 1 AND 2 AND 4.

Using DO in this direct way, we can print the result of evaluating any output of PREFIX+-:

```
←DO SENTENCE "PRINT" AND PREFIX+- "123456789+987654321+5555555555"
6666666665
```

Often, however, we will want to use the result of evaluation of the given infix expression as an input for further procedures. For example, we might want to test whether two infix expressions give the same result by evaluating each of them and comparing the results. To do this we need to OUTPUT the result of evaluation, not just PRINT it. We, therefore, write a procedure COMPUTE /EXP/ which outputs the result of evaluating PREFIX+- /EXP/. This procedure requires only one instruction line -- instead of DO SENTENCE "PRINT" PREFIX+- /EXP/, we use DO SENTENCE "OUTPUT" PREFIX+- /EXP/.

```
TO COMPUTE /EXP/
10 DO SENTENCE "OUTPUT" PREFIX+- /EXP/
END

←PRINT COMPUTE "123456789+987654321+5555555555"
6666666665

←TEST IS COMPUTE "1+9+2+8" COMPUTE "3+7+4+6"
←IF TRUE PRINT "YES"
YES
```

It is also useful for our further translation work to have a procedure, PERFORM, more general than COMPUTE, which outputs the effect of DO on its input, rather than the effect of DO on PREFIX+- applied to the input.

```
TO PERFORM /EXP/
10 DO SENTENCE "OUTPUT" /EXP/
END
```

To use this procedure to evaluate infix expressions, one can write:

```
←PRINT PERFORM PREFIX+- "1-1-1"
-1
```

PERFORM can, of course, be used without PREFIX+-

```
←PRINT PERFORM "PRINT 123"
123

←PRINT PERFORM PREFIX "1+2+3+4+5"
15
```

It is of interest to note that we could have produced a result somewhere between the *generation* of a prefix expression (by PREFIX+-) and its *evaluation* (by COMPUTE). Instead, we could have generated a LOGO *procedure for evaluating* the expression. This procedure could then be used at any time subsequently to carry out the numerical evaluation.

Let us do this. We will write the two-input procedure GENERATE /NAME/ /EXP/ which is to generate a LOGO procedure with the name /NAME/ for evaluating the infix expression /EXP/.

```
←TO GENERATE /NAME/ /EXP/
>10 DO SENTENCE OF "TO" AND /NAME/        (This generates the
>20 DO SENTENCE OF SENTENCE OF             title line)
     "10" AND "OUTPUT" AND PREFIX+- /EXP/ (This generates the line
                                          10 OUTPUT PREFIX+- /EXP/)
>30 DO "END"                            (This generates the END line)
>END
←
```

Let us try GENERATE with one of the examples used above:

```
←GENERATE "EXAMPLE" "1+2-4"
←
```

Let's check to see that GENERATE actually created a new procedure called EXAMPLE, by trying to list it.

```
←LIST EXAMPLE
TO EXAMPLE
10 OUTPUT DIFFERENCE OF SUM OF 1 AND 2 AND 4
END
←
```

When we execute this procedure it evaluates the prefix expression:

```
←PRINT EXAMPLE
-1
←
```

Apart from its use in subsequent translation tasks, GENERATE is of interest in illustrating (in a simple situation) how we can make procedures which make procedures. This kind of process fascinates children and also has important applications in programming language translation.

2.6 An Infix-Prefix Translation Quiz

Now that we have written a set of procedures which can translate infix expressions containing + and - signs in any combination, what shall we use them for? After such a large amount of work, the idea of a test seems appropriate and since it is far more fun to make up tests than to take them, let us write a testing procedure, which administers infix-prefix translation tests. One approach is to have the computer give the student an infix expression, ask him to translate it to prefix, and compare the answer to the one that the computer gets using PREFIX+-. We can also use COMPUTE to compute the numerical value of any student response which doesn't exactly match that of the computer and see if it gives the same number as the computer-derived expression. If so, the student response may be correct, though not identical in form to the computer-generated expression.

First, how should problems be given to the student? One possibility is to prepare a list of infix expressions for him to translate and assign these one at a time. The problems and associated solutions (equivalent prefix expressions) could be stored. But, this is inelegant. Using PREFIX+-, we can do a lot better. In particular, a much nicer idea is to have the computer randomly generate its own infix expressions. How can we do this? To begin with, we know that the computer can generate random digits, using the RANDOM procedure. This can be extended to the generation of random numbers of any size. A procedure to do this is shown in the appendix. Using this procedure, we can write a procedure to choose at random a member of an input list, /LIST/. (This procedure RND-CHOOSE /LIST/ is also described in the appendix.) Using this last procedure, it is easy to write a procedure for generating infix expressions composed of randomly chosen operators and operands, i.e., a random infix expression generator.

Let us first create two lists, one containing operands and the other containing operators. These lists will be used as inputs to RND-CHOOSE.

```
←MAKE
    NAME:  "OPERATORS"
    THING:  "+-"                  (So far we can only translate
←MAKE                              +, -)
    NAME:  "OPERANDS"
    THING:  "1 2 3 4 5 6 7 8 9"   (For simplicity, let's just use
                                  the non-zero digits for our
                                  operands)
```

Now we can create infix expressions:

```
←TO RND-INFIX
>10 OUTPUT SENTENCE OF SENTENCE OF
    RND-CHOOSE /OPERANDS/
    RND-CHOOSE /OPERATORS/
    RND-CHOOSE /OPERANDS/
>END

←PRINT RND-INFIX
4 - 3
```

To get longer expressions, we modify RND-INFIX so it generates an expression with /N/ operators.

```
←TO RND-INFIX /N/
>10 TEST IS /N/ 0
>20 IF TRUE OUTPUT RND-CHOOSE /OPERANDS/
>30 OUTPUT SENTENCE OF SENTENCE OF
     RND-CHOOSE /OPERANDS/
     RND-CHOOSE /OPERATORS/
     RND-INFIX (DIFFERENCE OF /N/ AND 1)
>END
```

For example:

```
←PRINT RND-INFIX 4
5-4+7+2-7
```

It is easy to write a procedure TRYOUT /N/ which tests, not only the RND-INFIX procedure we have just written, but also all the other procedures we need for our quiz.

```
←TO TRYOUT /N/
>10 SPACE 1                    (The procedure SPACE /N/ types /N/
                                empty lines)
>20 MAKE
     NAME:  "PROBLEM"
     THING:  RND-INFIX /N/
>30 PRINT /PROBLEM/            (This prints out the infix
                                expression to be translated)
>40 SPACE 2
>50 MAKE
     NAME:  "SOLUTION"
     THING:  PREFIX+- /PROBLEM/
>60 PRINT /SOLUTION/           (This prints out correct prefix
                                expression translation of the infix
                                expression given as a problem)
>70 SPACE 2
>80 PRINT COMPUTE OF /SOLUTION/  (This evaluates the prefix
                                  expression and prints the result)
>END
```

```
←TO SPACE /N/
>10 TEST IS /N/ 0
>20 IF TRUE STOP
>30 PRINT /EMPTY/                  (The typewriter skips a line)
>40 SPACE (DIFFERENCE OF /N/ AND 1)
>END
```

Now let's use TRYOUT for a moderate-sized expression.

```
←TRYOUT 5

3 + 7 - 4 - 7 + 4 - 5


DIFFERENCE OF SUM OF DIFFERENCE OF DIFFERENCE OF SUM OF 3 AND 7 AND 4
AND 7 AND 4 AND 5


-2
```

We can as easily generate *much* harder-looking problems.

```
←TRYOUT 25

7 + 5 - 3 - 9 + 2 + 8 - 3 + 2 + 2 - 8 + 3 - 8 - 7 - 9 - 4 - 1 + 2 + 7 -
9 - 5 + 5 - 1 + 4 + 8 + 7 + 6


SUM OF SUM OF SUM OF SUM OF DIFFERENCE OF SUM OF DIFFERENCE OF
DIFFERENCE OF SUM OF SUM OF DIFFERENCE OF DIFFERENCE OF DIFFERENCE OF
DIFFERENCE OF DIFFERENCE OF SUM OF DIFFERENCE OF SUM OF SUM OF
DIFFERENCE OF SUM OF SUM OF DIFFERENCE OF DIFFERENCE OF SUM OF 7 AND 5
AND 3 AND 9 AND 2 AND 8 AND 3 AND 2 AND 2 AND 8 AND 3 AND 8 AND 7 AND 9
AND 4 AND 1 AND 2 AND 7 AND 9 AND 5 AND 5 AND 1 AND 4 AND 8 AND 7 AND 6


1
```

At this point, we are ready to write our basic test procedure.

```
←TO PREFIX-TEST /N/
>10 MAKE "PROBLEM" RND-INFIX /N/          (Generate a random infix
>20 PRINT SENTENCE OF SENTENCE OF          expression)
     "TRANSLATE THE INFIX EXPRESSION:"    (Use it as a problem)
     /PROBLEM/
     "INTO PREFIX FORM"
>30 MAKE "STUDENTANS" REQUEST             (Get the student's answer)
>40 MAKE "COMPUTERANS" PREFIX+- /PROBLEM/ (Get a correct answer)
>50 TEST IS /STUDENTANS/ /COMPUTERANS/    (Compare them)
>60 IF TRUE PRINT "RIGHT"                 (If they are identical,
>70 IF TRUE STOP                           indicate this and stop)
>80 TEST IS COMPUTE /STUDENTANS/ COMPUTE /COMPUTERANS/ (If they give
>90 IF TRUE PRINT "YOUR ANSWER EVALUATES TO THE SAME   same result,
     NUMBER AS MINE, BUT IS DIFFERENT IN FORM"          say so)
>100 IF FALSE PRINT SENTENCE OF                (Else, give student
     "WRONG, A CORRECT ANSWER IS:"             our answer)
     /COMPUTERANS/
>END
←
```

Instead of requiring the user to type in PREFIX-TEST and an argument each time he takes the test, we could have another procedure which repeats the procedure PREFIX-TEST as often as desired:

```
←TO PREF-TEST
>10 PRINT "HOW MANY OPERATORS DO YOU WANT IN THE PROBLEM?"
>20 PREFIX-TEST REQUEST
>30 PRINT "ANOTHER PROBLEM?"
>40 TEST IS REQUEST "YES"
>50 IF TRUE GO TO LINE 10
>60 PRINT "GOODBYE THEN"
>END
←
```

Let's try the test procedure now.

```
←PREF-TEST
HOW MANY OPERATORS DO YOU WANT IN THE PROBLEM?
::3
TRANSLATE THE INFIX EXPRESSION: 1+4-9+3 INTO PREFIX FORM
::SUM OF DIFFERENCE OF SUM OF 1 AND 4 AND 9 AND 3
RIGHT
ANOTHER PROBLEM?
::YES
   .
   .
   .
```

This basic test procedure can be embellished considerably. For example, the computer can do the usual things like keep track of the number of right and wrong answers, and the response times, print a score at the end of the session, etc. The computer could compute the number of operators to be used in the next question, perhaps by increasing the number of operators if the answer is correct and decreasing them if the answer is wrong. Also, in the case of an incorrect student answer, the computer could find the part of the answer in which the error lies and try to help the student appropriately. Instead of going in these directions, however, we shall return to our main path and continue with the translation problem.

2.7 Completing the Infix-Prefix Translator

Two extensions of our infix-prefix translator remain to be made. The conversion of * and / (multiplication and division) operations from infix to appropriate prefix form must be added. We also wish to include the ability to handle expressions with parentheses. Incorporating * and / should be as straightforward as the inclusion of - translation. The translation of expressions with parentheses will present some problems, since, as we have already seen, the present version of PREFIX+- handles parentheses very badly.

```
←PRINT PREFIX+- "1-(2-3)"
DIFFERENCE OF DIFFERENCE OF 1 AND (2 AND 3)
```

Let us first deal with * and /. Before we start thinking about the actual procedures we are going to write, let us recall the properties of multiplication and division. First, and this is essential in the evaluation of any expression, the multiplications and divisions are to be done first, before additions and subtractions. Thus, in writing a *, / translation procedure, we look for additions and subtractions first (recalling that the order of evaluation is the opposite of the order of parsing) and only after all these have been found do we look for * and / symbols. The translation only begins once we have reached the end of our searching -- this means, as it did earlier with + and -, that the * and / will actually be translated first. We say, also, that divisions and multiplications can be performed left to right to give correct results. This means that we want to search for the rightmost * and /, then the next, and so on, starting to translate only when we reach the last one. Thus we have very much the same situation with * and / as we had with + and -.

Moreover, the analogy goes even further, since, in fact, multiplications can be performed in any order, just as additions can; divisions, on the other hand, like subtractions, cannot. 2*3*4 is 24, whether the 2 and 3 are multiplied first or the 3 and 4 are multiplied first. Consider, however, 6/3/2. If the division 3/2 were performed first, the answer would be 4. Using the standard convention going from left to right, the division 6/3 is done first and the final result is 1. The fact that multiplication is associative can be used to simplify the treatment of * and / just as the treatment of + and - was simplified by use of the associativity of +.

Let us call the procedure which translates all four basic operations PREFIX. Our strategy in extending PREFIX+- to obtain PREFIX, is to look for all * symbols after we have looked for + and - signs. After this is done, we look for all / symbols. If none of these appear, we simply output the input expression. Our last version of PREFIX+- was:

```
TO PREFIX +- /EXP/
10 TEST IS (RIGHT "+" /EXP/) /EXP/
20 IF FALSE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
     "SUM OF"
     PREFIX+- LEFT "+" /EXP/
     "AND"
     PREFIX+- RIGHT "+" /EXP/
30 TEST IS (RIGHT "-" /EXP/) /EXP/
40 IF FALSE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
     "DIFFERENCE OF"
     PREFIX+- LEFT "-" /EXP/
     "AND"
     PREFIX+- RIGHT "-" /EXP/
50 OUTPUT /EXP/
END
```

To accomplish the changes described in our discussion of strategy, we change the name PREFIX+- everywhere it appears to PREFIX. We eliminate line 50 and add these five lines:

```
50 TEST IS (RIGHT "*" /EXP/) /EXP/
60 IF FALSE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
     "PRODUCT OF"
     PREFIX LEFT "*" /EXP/
     "AND"
     PREFIX RIGHT "*" /EXP/
70 TEST IS (RIGHT "/" /EXP/) /EXP/
80 IF FALSE OUTPUT SENTENCE OF SENTENCE OF SENTENCE OF
     "QUOTIENT OF"
     PREFIX LEFT "/" /EXP/
     "AND"
     PREFIX RIGHT "/" /EXP/
90 OUTPUT /EXP/
END
```

and we are done with the first part of our generalization of PREFIX+-.

Let us try out our new procedure:

```
←PRINT PREFIX "2*3-5/6*7"
DIFFERENCE OF PRODUCT OF 2 AND 3 AND PRODUCT OF QUOTIENT OF 5 AND
6 AND 7
```

We turn, now, to the harder problem of inclusion of parenthesized expressions in our infix repertoire. An important simplification in this problem is that no parentheses are needed to represent any arbitrary sequence of operations in prefix notation. This is not an obvious point. It is most easily seen by considering how expressions are built up in infix and in prefix notation. Say we start with the sum of 2 and 3 -- in infix, 2+3. How can we write the product of this sum and 5? We cannot do it without parentheses since multiplications are otherwise always performed before additions and thus we must write 5*(2+3). The *prefix* expression for the sum is SUM OF 2 AND 3. To indicate the product of this sum and 5, we write PRODUCT OF 5 AND SUM OF 2 AND 3. More generally, as one builds up prefix expressions, the operation last added is the one last to be performed. Prefix expressions, therefore, can only be evaluated in the opposite order to which they were built up, regardless of the operations involved. There is no need for parentheses here, or for any left-right conventions, as there is with infix notation.

Keeping in mind that parentheses are superfluous in prefix expressions, we consider the translation of infix expressions containing parentheses. To discuss the problem concretely, we start with the example discussed previously, 5*(2+3). If we translate the part of this expression within parentheses into prefix notation separately as (SUM OF 2 AND 3), the parentheses become unnecessary, so let's remove them. We then have left the translation of the expression 5* SUM OF 2 AND 3 and this is PRODUCT OF 5 AND SUM OF 2 AND 3. Our strategy for translation, therefore, is to translate the contents of the innermost parentheses first, making that set of parentheses superfluous and, by continuing this process to eliminate all parentheses, translating the contents of each one along the way. (See Footnote next page.)

This process of finding the innermost set of parentheses is really more complicated than the apparently similar recursion used with operators. The reason for this is that in cases like 2-(3*4-5)*6+7, one has to keep track, not only of the contents of the parentheses, but also of what is to both the left and right of them. A simple way of side-stepping this complicated recursion is as follows. When we come to the first parenthesis, replace the part of the expression between it and the matching parenthesis by, say, $1, then, the contents of the next set by $2, and so on, assigning each of these as the name of the contents. (We use the symbol $ to avoid ambiguities since it is not used elsewhere in expressions.)

For example, take the complicated expression (2-3)*(4+5/(6+7)). We come to the first pair of parentheses and replace their contents by $1. Our expression now looks like $1*(4+5/(6+7)). We come to the next parenthesis, and by counting find that the matching parenthesis is the rightmost one. We replace their

---

Footnote: Faced with a complicated expression like ((3+4/7)*(9/(1-5)-(7-3))), it is not obvious how the parentheses are associated, for example, which right parenthesis pairs with the leftmost one. A good algorithm is to count parentheses, adding 1 each time a left parenthesis is met, and subtracting one each time a right parenthesis is encountered. If one starts by naming the given parenthesis 1, then the matching parenthesis is the first one which has the number 0. In the example above,

```
( ( 3+4/7 ) * ( 9/ ( 1-5 ) - ( 7-3 ) ) )
1 2       1   2    3     2   3     2 1 0
```

Thus, in this case, the right parenthesis matching the leftmost one is the rightmost one. One way of seeing why this works is to observe that in any valid mathematical expression, the parentheses must balance -- that is, there must be an equal number of left and right parentheses. Thus, in our counting process, we are simply finding the parenthesis which, with the given one, and the portion of the expression in between, forms a valid arithmetic expression. This subexpression, as any operand, *must* itself be a valid arithmetic expression.

contents by \$2. Our expression now looks like \$1*\$2 where \$1 is the name of 2-3 and \$2 is the name of 4+5/(6+7). We replace the part of \$2 within parentheses, 6+7, by \$3. We can now summarize the situation as follows:

original expression is \$1*\$2
\$1 names 2-3
\$2 names 4+5/\$3
\$3 names 6+7

There are no longer any parentheses, either in our original expression or in any of the subexpressions, so we can now translate them easily into prefix. Let us do this.

original expression: PRODUCT OF \$1 AND \$2
\$1 DIFFERENCE OF 2 AND 3
\$2 SUM OF 4 AND QUOTIENT OF 5 AND \$3
\$3 SUM OF 6 AND 7

We now can insert back all the subexpressions into the original. First we substitute \$3 back into \$2, getting for \$2 SUM OF 4 AND QUOTIENT OF 5 AND SUM OF 6 AND 7. Next we substitute \$1 and \$2 into the original expression, obtaining

PRODUCT OF DIFFERENCE OF 2 AND 3 AND SUM OF 4 AND QUOTIENT OF 5 AND SUM OF 6 AND 7

The task of writing computer procedures to carry out the method described above falls naturally into two parts. First, the contents of parentheses are replaced by names of the form \$I, where I starts at 1 and is increased by 1 at each subsequent naming. When all parentheses have been replaced, all of the subexpressions can be translated into prefix notation. The second part of the general method is the substitution of all subexpressions back into the original expression.

The procedure used for the first part of the process is called INSIDE. Given an expression /EXP/ containing a right parenthesis and a number /PCOUNT/ initially zero, INSIDE keeps looking through /EXP/ until it finds the matching parenthesis. INSIDE then outputs the contents of the pair of parentheses. It also gives the name "LEFTSIDE" to the part of /EXP/ to the left of the matching parenthesis. Thus,

```
←PRINT INSIDE "1+(2-3)"
2-3
←PRINT /LEFTSIDE/
1+
```

We now want to modify PREFIX to make use of INSIDE. We have to augment PREFIX so that before it does anything else in the way of translation, it looks for parentheses. If it finds a right parenthesis, it makes up a new name by adding 1 to the value of I and putting $ in front. PREFIX gives this name to the output of INSIDE /EXP/, which is the part of /EXP/ between the first right parenthesis and its matching parenthesis. This subexpression is replaced by its name in /EXP/. PREFIX is then applied to the new /EXP/. If PREFIX does not find any parentheses, it translates the expression /EXP/ into prefix notation, just as before.

This completes the first part of our translation procedure. FILL performs the substitutions of subexpressions back into the original expression. It looks for anything of the form $I and replaces all such objects by FILL of the THING that they name. By doing this recursively, we ensure that all occurrences of $I will be replaced even if they occur in subexpressions themselves. Thus, with this change in PREFIX, and the new procedures INSIDE and FILL, our general method is implemented. We need only add the starting procedure, which we call PREFIXALL, which starts off I with the value 0 and then applies FILL to the output of PREFIX. These procedures, for the complete infix-prefix translation process, are listed next.

```
TO PREFIX /EXP/
1 TEST IS RIGHTS ")" /EXP/ /EXP/   (Is there any ")"?)
2 LOCAL "NAME" "LEFTSIDE"  (The command LOCAL is discussed below)
3 IF FALSE MAKE "I" SUM /I/ "1"          (If there is a ")")
4 IF FALSE MAKE "NAME" WORD "$" /I/
5 IF FALSE MAKE /NAME/ PREFIX INSIDE /EXP/ "0"
6 IF FALSE OUTPUT PREFIX WORD WORD
     /LEFTSIDE/
     /NAME/
     RIGHTS ")" /EXP/
10 TEST IS RIGHTS "+" /EXP/ /EXP/
20 IF FALSE OUTPUT SENTENCE SENTENCE SENTENCE
     "SUM OF"
     PREFIX LEFTS "+" /EXP/
     "AND"
     PREFIX RIGHTS "+" /EXP/
30 TEST IS RIGHTS "-" /EXP/ /EXP/
40 IF FALSE OUTPUT SENTENCE SENTENCE SENTENCE
     "DIFFERENCE OF"
     PREFIX LEFTS "-" /EXP/
     "AND"
     PREFIX RIGHTS "-" /EXP/
50 TEST IS RIGHTS "*" /EXP/ /EXP/
60 IF FALSE OUTPUT SENTENCE SENTENCE SENTENCE
     "PRODUCT OF"
     PREFIX LEFTS "*" /EXP/
     "AND"
     PREFIX RIGHTS "*" /EXP/
70 TEST IS RIGHTS "/" /EXP/ /EXP/
80 IF FALSE OUTPUT SENTENCE SENTENCE SENTENCE
     "QUOTIENT OF"
     PREFIX LEFTS "/" /EXP/
     "AND"
     PREFIX RIGHTS "/" /EXP/
90 OUTPUT /EXP/
END
```

```
TO RIGHTS /SYMBOL/ /EXP/
10 TEST IS /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 TEST IS LAST /EXP/ /SYMBOL/
40 IF TRUE OUTPUT /EMPTY/
50 TEST WORDP /EXP/
60 IF TRUE OUTPUT WORD
     RIGHTS /SYMBOL/ BUTLAST /EXP/
     LAST /EXP/
70 OUTPUT SENTENCE
     RIGHTS /SYMBOL/ BUTLAST /EXP/
     LAST /EXP/
END
```

```
TO LEFTS /SYMBOL/ /EXP/
10 TEST IS /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 TEST IS LAST /EXP/ /SYMBOL/
40 IF TRUE OUTPUT BUTLAST /EXP/
50 OUTPUT LEFTS /SYMBOL/ BUTLAST /EXP/
END


TO INSIDE /EX/ AND /PCOUNT/
10 TEST BOTH ZEROP /PCOUNT/ IS LAST OF /EX/ ")"    (First ")"?)
20 IF TRUE OUTPUT INSIDE OF BUTLAST OF /EX/ AND "1" (If true add 1
                                                  to parenthesis count
                                                  and keep looking)
30 TEST ZEROP /PCOUNT/                     (Not yet come to first ")")
40 IF TRUE OUTPUT INSIDE OF BUTLAST OF /EX/ AND "0"
50 TEST IS LAST OF /EX/ ")"
60 IF TRUE OUTPUT WORD OF
      INSIDE OF BUTLAST OF /EX/ AND SUM OF /PCOUNT/ AND "1"
      ")"                                 (Another ")" add 1 to count)
70 TEST IS LAST OF /EX/ "("              (If "(" subtract 1 from count)
80 IF TRUE MAKE "PCOUNT" DIFFERENCE OF /PCOUNT/ AND "1"
90 IF TRUE TEST ZEROP /PCOUNT/                    (Are we finished?)
95 IF TRUE MAKE "LEFTSIDE" BUTLAST OF /EX/}
100 IF TRUE OUTPUT /EMPTY/               }(If we are finished)
110 OUTPUT WORD OF
      INSIDE OF BUTLAST OF /EX/ AND /PCOUNT/
      LAST OF /EX/
END


TO FILL /SEN/
10 TEST EMPTYP /SEN/ (If /SEN/ is /EMPTY/ it is time to stop)
20 IF TRUE OUTPUT /EMPTY/
25 TEST WORDP /SEN/                     (If input is a word does it)
26 IF TRUE TEST IS FIRST /SEN/ "$"  (begin with $                )
27 IF TRUE OUTPUT FILL OF THING OF /SEN/ (If so FILL thing of input)
30 TEST IS FIRST OF FIRST OF /SEN/ "$"   (Does first word of input
                                          begin with $)
40 IF TRUE OUTPUT SENTENCE
      FILL OF THING OF FIRST OF /SEN/
      FILL BUTFIRST /SEN/
50 OUTPUT SENTENCE
      FIRST OF /SEN/                     (If no $ appeared, try
      FILL BUTFIRST /SEN/                 FILL on BUTFIRST /SEN/)
END

TO PREFIXALL /EXP/
5 MAKE "I" "0"
10 OUTPUT FILL PREFIX /EXP/
END
```

It should be noted that we are assuming that the input expression /EXP/ is a *word*, following the usual practice of writing parentheses right next to the operands of an arithmetic expression, rather than with spaces on either side. The use of PREFIXALL in translating a parenthesized expression is illustrated by a TRACE in the appendix.

The use of the LOGO command LOCAL, which occurs in PREFIX, is described next. Assume that several procedures all use a name, say "NAME", which is not an input of any of them. Then each time that "NAME" is given a new THING, the THING OF "NAME", is changed for *all* these procedures. All previous THINGs of "NAME" are forgotten. This is the usual situation with names that are not inputs to procedures. We might say such names are *global* or *public*, since their things are the same for all procedures using them. Sometimes, however, it is desirable to use a word, such as "NAME" as a name for a different thing in each procedure where it is used. The command LOCAL "NAME" does this. LOCAL causes its inputs to be treated as "private" to each procedure in which the command is used. LOCAL is especially useful when a procedure calls itself recursively and we want some name to have a different THING at each level of recursion. Consider a procedure which is supposed to reverse its input word, i.e., to "write it backwards".

```
TO REVERSE /INPUT/
10 TEST IS /INPUT/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 MAKE "Y" FIRST OF /INPUT/
40 OUTPUT WORD OF
     REVERSE (BUTFIRST OF /INPUT/) AND /Y/
END

←PRINT REVERSE "HELLO"
OOOOO
```

172

Let's see what REVERSE does so we understand why it didn't work.

```
TRACE REVERSE
REVERSE "HELLO"
REVERSE OF "HELLO"                (/Y/ is now "H")
  REVERSE OF "ELLO"               (/Y/ is now "E")
    REVERSE OF "LLO"              (/Y/ is now "L")
      REVERSE OF "LO"             (/Y/ is now "L")
        REVERSE OF "O"            (/Y/ is now "O")
          REVERSE OF ""   (Since input is /EMPTY/, /Y/ is not changed)
          REVERSE OUTPUTS ""
        REVERSE OUTPUTS "O"
      REVERSE OUTPUTS"OO"
    REVERSE OUTPUTS "OOO"
  REVERSE OUTPUTS "OOOO"
REVERSE OUTPUTS "OOOOO"
```

"Y" is made "O" in the innermost REVERSE and never changes thereafter. In the succeeding outputs, this /Y/ is what is actually used. We really intended that the THING that "Y" was given on each level would be preserved for that procedure no matter how "Y" would change subsequently. We can easily accomplish this by the insertion of the instruction:

```
5 LOCAL "Y"
```

Now REVERSE works.

```
←PRINT REVERSE "HELLO"
OLLEH
```

* * * * * * * * *

PREFIX is finished. We could next turn things around and develop the procedure INFIX for prefix to infix translation. We will do so in time, but first we would rather go straight on to deal with story problems. Once we can translate prefix expressions into story expressions, we will be able to utilize the work we have just finished on translating infix to prefix and go from relatively formal infix equations to relatively informal story problems.

3. The Translation of Simple Story Problems: Infix Expressions to Story Expressions

The theme underlying all the work in this unit is the gradual development of methods to transform formal equations, in infix form, to what we call *story problems* (and vice versa). The translation of infix to prefix form is an important part of this process. Certainly, the phrase "sum of John's age and 18" is much closer to everyday English speech, than is the equivalent infix "John's age + 18".

Let us now start on our development of story problems from infix expressions. Obtaining "sum of John's age and 18" from "John's age + 18" seems to be a good first step, though there is still a long way to go. Even better, as story problems go, is the form: "John's age 18 years from now". An apparent difficulty in this kind of translation is that each type of story problem requires a different type of translation from the prefix form. Clearly the translation used in the example preceding, "sum of /PERSON/'s age and /NUMBER/" to "/PERSON/'s age /NUMBER/ years from now" is only useful for age problems. Problems involving, say distance, would require a different translation, for example, "sum of distance to /PLACE/ and /NUMBER/" to "/NUMBER/ miles farther than /PLACE/". Thus we have special, but fixed, patterns to translate for each standard type of problem we want to do. So we need a set of procedures for handling a rich diversity of patterns.

3.1 Substitution

Our general approach is to replace certain patterns in a prefix expression by certain other patterns. We will eventually reach the point of developing relatively powerful substitution capabilities that will allow us to generate a large class of simple story problems from infix expressions. Let us start, however, in a modest way by writing a procedure for replacing one character by another.

In this simple substitution procedure we shall replace all occurrences of a given character, /CHAR/, by another character or word, /SUBST/, in a given expression, /EXP/. We shall write a procedure REPLACE /CHAR/ /SUBST/ /EXP/ to do this simple replacement. REPLACE searches through /EXP/ one character at a time. We compare the first character of /EXP/ with /CHAR/. If they match, we output the word formed from /SUBST/ and REPLACE of the rest of /EXP/. If they do not match, we apply REPLACE to the rest of /EXP/. Thus,

```
←TO REPLACE /CHAR/ /SUBST/ /EXP/
>10 TEST IS /EXP/ /EMPTY/                 (If /EXP/ is /EMPTY/ we
>20 IF TRUE OUTPUT /EMPTY/                 are finished)
>30 TEST IS FIRST OF /EXP/ /CHAR/
>40 IF TRUE OUTPUT WORD OF                (Replace first character
     /SUBST/                               by /SUBST/ and continue
     REPLACE /CHAR/ /SUBST/ (BUTFIRST OF /EXP/)  the process)
>50 OUTPUT WORD OF                        (Leave first character
     FIRST OF /EXP/                        of /EXP/ unchanged and
     REPLACE /CHAR/ /SUBST/ (BUTFIRST OF /EXP/) continue the process)
>END
```

```
←PRINT REPLACE "A" "Z" "ALGEBRA"
ZLGEBRZ
```

We can also use REPLACE to substitute a word for a character:

```
←PRINT REPLACE "V" "ATH" "LOVE"
LOATHE
```

Since we need to work with *sentences* in story problems, we shall modify REPLACE so that /EXP/ is a sentence, rather than a word. Looking back at the REPLACE we have just written, we see that the only places in which a distinction between WORD and SENTENCE is used, is in the two output lines 40 and 50. If we change WORD to SENTENCE in these lines, then the procedure will treat the input /EXP/ as a sentence. Alternatively we can use the operation WORDP /EXP/ to test whether or not /EXP/ is a word. Then the operation WORD would be used in the formation of the output when /EXP/ is a word; otherwise, the operation SENTENCE would be used. After we modify REPLACE in either of these ways, we can translate such things as:

```
←PRINT REPLACE "X" "JACK'S AGE" "SUM OF X AND 3"
SUM OF JACK'S AGE AND 3
```

This is closer to the sort of translation we wish to perform, but it is still too limited. It can only deal with patterns consisting of a single word. Thus, we can do:

```
←PRINT REPLACE "TWICE" "PRODUCT OF 2 AND" "TWICE 2"
PRODUCT OF 2 AND 2
```

But we cannot go the other way and write

```
REPLACE "PRODUCT OF 2 AND" "TWICE" "TWICE 2"
```

since the first input of REPLACE needs to be a word. We need the ability to replace any part of an expression. So we shall extend REPLACE to do this. We can split this more complicated problem into two parts.

We first write a procedure FIND /LIST/ /EXP/ which finds out whether or not /LIST/ is a starting subexpression of /EXP/. (For example, "NOW IS" is a starting subexpression of "NOW IS THE TIME".) If not, FIND outputs "NO MATCH". If /LIST/ does form the first part of /EXP/, FIND outputs that part of /EXP/ which follows /LIST/.

```
TO FIND /LIST/ /EXP/
10 TEST IS /LIST/ /EMPTY/          (Has all of /LIST/ already been
20 IF TRUE OUTPUT /EXP/             matched?)
30 TEST IS /EXP/ /EMPTY/           (Have we gone through all of
                                    /EXP/ unsuccessfully?)
40 IF TRUE OUTPUT "NO MATCH"
50 TEST IS FIRST /LIST/ FIRST /EXP/
60 IF TRUE OUTPUT FIND (BUTFIRST /LIST/) (If the first members of
   (BUTFIRST /EXP/)                      /LIST/ and /EXP/ match,
                                         go on to compare the next
70 OUTPUT "NO MATCH"                     element of each)
END
```

```
←PRINT FIND "NOW IS" "NOW IS THE TIME"
THE TIME
←PRINT FIND "A B" "C A B C A B"
NO MATCH
←PRINT FIND "A B" "A B C A B C"
C A B C
```

The main procedure, our new REPLACE, will apply FIND to the expression to be translated, /EXP/. If the output of FIND is "NO MATCH", REPLACE tries again with BUTFIRST of the expression to be translated. If the output of FIND is something else (i.e., *not* "NO MATCH"), REPLACE replaces /LIST/ by /SUBSTLIST/ in /EXP/ and continues to look for further possible substitutions in the rest of /EXP/. REPLACE makes the desired substitution simply by combining /SUBSTLIST/ and the output of FIND. (The output of FIND is precisely the rest of /EXP/, that is the part of /EXP/ following the match.)

```
TO REPLACE /LIST/ /SUBSTLIST/ /EXP/
10 TEST IS /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 MAKE
     NAME:  "REST OF EXP"            (The output of FIND is the part
     THING:  FIND /LIST/ /EXP/       of /EXP/ following /LIST/)
40 TEST IS /REST OF EXP/ "NO MATCH"   (If there was no match, try
50 IF TRUE OUTPUT SENTENCE OF         for a match with the rest
   FIRST /EXP/                        of /EXP/)
   REPLACE /LIST/ /SUBSTLIST/ BUTFIRST /EXP/
```

(continued)

```
60 OUTPUT SENTENCE OF            (When a match is found, substitute
    /SUBSTLIST/                  /SUBSTLIST/ and repeat process on
    REPLACE /LIST/ /SUBSTLIST/ /REST OF EXP/   rest of /EXP/)
END
```

Using this new version of REPLACE, we can make substitutions which look very much like the kind we are after:

```
←PRINT REPLACE "SUM OF JACK'S AGE AND 3" "JACK'S AGE 3 YEARS
  FROM NOW" "SUM OF 4 AND SUM OF JACK'S AGE AND 3 EQUALS 20"
SUM OF 4 AND JACK'S AGE 3 YEARS FROM NOW EQUALS 20
```

Although the translation in this specific case is just what we want, the REPLACE procedure would not have been so successful for "SUM OF 4 AND SUM OF JACK'S AGE AND 2 EQUALS 20", even though this expression is almost the same as the other.

We would like to have a version of REPLACE which would work for SUM OF anyone's AGE AND anynumber. A good way to get this kind of generalization is to regard all words in /LIST/ starting with the three letters A N Y as matching *any words whatever* in /EXP/. For example, the words "ANYOLD AGE" appearing in /LIST/ would be taken as matching the words "JACK'S AGE" appearing in /EXP/. After matching, we want to record that "JACK'S" matched "ANYOLD" so we can later replace /ANYOLD/ by "JACK'S" in carrying out the translation. As another example, the list "ANY1 + ANY2" would match "13 + 27" in /EXP/ and /ANY1/ would be made "13" while /ANY2/ would be made "27".

At present, the only condition under which FIND will go on to examine the next character of /LIST/ and /EXP/ is if the first members of the two are identical. The new condition, that the first word of /LIST/ starts with the letters A N Y, leads to the same effect. We need to add this condition to FIND. An auxiliary procedure F3, to output the first three characters of a word, is useful for modifying FIND.

```
TO F3 /WORD/
10 OUTPUT WORD OF WORD OF
     FIRST /WORD/                             (First character)
     FIRST BUTFIRST /WORD/                    (Second character)
     FIRST BUTFIRST BUTFIRST /WORD/           (Third character)
END
```

Then the steps to be added to FIND are 63, 65, 67, underlined in the following. All other steps are unchanged.

```
TO FIND /LIST/ /EXP/
10 TEST IS /LIST/ /EMPTY/
20 IF TRUE OUTPUT /EXP/
30 TEST IS /EXP/ /EMPTY/
40 IF TRUE OUTPUT "NO MATCH"
50 TEST IS (FIRST /LIST/) (FIRST /EXP/)
60 IF TRUE OUTPUT FIND (BUTFIRST /LIST/) (BUTFIRST /EXP/)
63 TEST IS F3 FIRST /LIST/ "ANY"   (First 3 letters of first word
65 IF TRUE MAKE (FIRST /LIST/) (FIRST /EXP/)  of /LIST/ = "ANY")
67 IF TRUE OUTPUT FIND (BUTFIRST /LIST/) (BUTFIRST /EXP/) (If so,
70 OUTPUT "NO MATCH"                            make the first word of
END                                            /LIST/ name the first
                                               word of /EXP/)
                                               (And continue searching)
```

```
←PRINT FIND "ANYOLD AGE" "SUM OF JACK'S AGE AND 3"
NO MATCH

←PRINT FIND "ANYOLD AGE" "JACK'S AGE AND 3"
AND 3

←PRINT /ANYOLD/
JACK'S
```

Now that we have incorporated this new capability for FIND to search for specified *patterns* in /EXP/, and to find and name the things in /EXP/ corresponding to the "ANY"-prefaced-words in these patterns, let's test it. Let's try REPLACE on a case where /LIST/ is a pattern of this kind.

```
←PRINT REPLACE "SUM OF ANYONE'S AGE AND ANYNUMBER" "ANYONE'S AGE
  ANYNUMBER YEARS FROM NOW" "SUM OF JACK'S AGE AND 4 EQUALS 15"
ANYONE'S AGE ANYNUMBER YEARS FROM NOW EQUALS 15
```

It didn't work. The expression to be substituted, "ANYONE'S AGE ANYNUMBER YEARS FROM NOW" was just copied in place of "SUM OF ANYONE'S AGE AND ANYNUMBER". "ANYONE'S" and "ANYNUMBER" were used in the result instead of /ANYONE'S/ and /ANYNUMBER/ (that is, "JACK'S" and "4"). Let's check to see that the two names were actually assigned correctly.

```
←PRINT /ANYONE'S/
JACK'S
←PRINT /ANYNUMBER/
4
```

The two names were correctly assigned but we forgot to substitute the things they name (instead of the names themselves) when we made the substitution of /SUBSTLIST/ for /LIST/ in REPLACE. To fix this and modify REPLACE appropriately, we need a procedure called ATTACH which replaces each word that begins with ANY by the THING of that word.

```
TO ATTACH /SEN/
10 TEST IS /SEN/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 TEST IS F3 FIRST /SEN/ "ANY"   (If word begins with "ANY")
40 IF TRUE OUTPUT SENTENCE        (Replace it with THING of the
     THING OF FIRST /SEN/          word and repeat the process)
     ATTACH BUTFIRST /SEN/
50 OUTPUT SENTENCE
     FIRST /SEN/                  (Otherwise, leave word unchanged
     ATTACH BUTFIRST /SEN/         and repeat the process)
END
```

Let's try ATTACH (with the same names used in the above example):

```
←PRINT ATTACH OF "ANYONE'S FACE HAS ANYNUMBER NOSES"
JACK'S FACE HAS 4 NOSES
```

We can now use ATTACH /SUBSTLIST/ instead of /SUBSTLIST/ when we perform the substitutions in REPLACE. To do this we change line 60 in REPLACE to:

```
60 OUTPUT SENTENCE OF
     ATTACH /SUBSTLIST/
     REPLACE /LIST/ /SUBSTLIST/ /REST OF EXP/
```

Now our REPLACE procedure works with patterns.

```
←PRINT REPLACE "SUM OF ANYONE'S AGE AND ANYNUMBER" "ANYONE'S AGE
  ANYNUMBER YEARS FROM NOW" "SUM OF JACK'S AGE AND 4 EQUALS 15"
JACK'S AGE 4 YEARS FROM NOW EQUALS 15
```

The procedure REPLACE is now sufficiently powerful to allow us to use some fairly general patterns to effect substitutions. For example, in age problems, as we saw, we can replace "SUM OF ANYONE'S AGE AND ANYNUMBER" by "ANYONE'S AGE ANYNUMBER YEARS FROM NOW" wherever the former pattern occurs in any given expression (attaching /ANYONE'S/ and /ANYNUMBER/ along the way).

To avoid the very considerable amount of typing necessary in specifying this replacement process, let us write a procedure AGEREPLACE, which transforms both the above form of sum and the corresponding difference:

```
TO AGEREPLACE /EXP/
10 MAKE "EXP" REPLACE "SUM OF ANYONE'S AGE AND ANYNUMBER"
   "ANYONE'S AGE ANYNUMBER YEARS FROM NOW" /EXP/
20 MAKE "EXP" REPLACE "DIFFERENCE OF ANYONE'S AGE AND ANYNUMBER"
   "ANYONE'S AGE ANYNUMBER  EARS AGO" /EXP/
30 OUTPUT /EXP/
END
```

This procedure is much more convenient to use than is REPLACE by itself.

```
←PRINT AGEREPLACE "SUM OF IRVING'S AGE AND 3"
IRVING'S AGE 3 YEARS FROM NOW

←PRINT AGEREPLACE "DIFFERENCE OF DAVID'S AGE AND 2 EQUALS 14"
DAVID'S AGE 2 YEARS AGO EQUALS 14
```

We do, however, run into difficulties when more complicated sorts of expressions are used. For example:

```
←PRINT AGEREPLACE "SUM OF JOHN'S AGE AND SUM OF 2 AND 2"
JOHN'S AGE SUM YEARS FROM NOW OF 2 AND 2
```

When REPLACE was used (in line 10 of AGEREPLACE) to match the pattern "SUM OF ANYONE'S AGE AND ANYNUMBER", it took the second "SUM" word in the input as the match for ANYNUMBER. Hence, the rather disconcerting result. When we specified the substitution for ANYNUMBER in line 10, we intended that ANYNUMBER should be matched by a *number*. We did not, however, put any such restriction into the program. Thus, REPLACE was free to substitute any word whatever for ANYNUMBER.

One solution to this difficulty is to explicitly write the *type* of word that the "ANY"-prefaced-word should replace in parentheses following it. For example, in the pattern "SUM OF ANYTHING(NUMBER) AND 1", the parentheses indicate that the word to be substituted for "ANYTHING" must be a number. We could then use age substitutions such as:

```
REPLACE "SUM OF ANYONE'S(NOTNUMBER) AGE AND ANY(NUMBER)"
"ANYONE'S(NOTNUMBER) AGE ANY(NUMBER) YEARS FROM NOW" /EXP/
```

To incorporate this notation, we must include a step in FIND which checks that the word that we are comparing with "ANY(   )" is of the right type. A first step in doing this is to write a procedure KINDP /ANY.../ /W/, where the first input begins with the letters A N Y, and the second input /W/ designates the word whose type we are testing. KINDP outputs TRUE if /W/ is of the form specified in the parentheses following the ANY, and FALSE otherwise. If there are no parentheses in the first input, KINDP will output TRUE. At least initially, let us restrict ourselves in KINDP to two types of words -- numbers and nonnumbers.

Later, if we wish, we can generalize this. The only change required in FIND to make use of KINDP is the addition of the line 64.

```
63 TEST IS F3 FIRST /LIST/ "ANY"
64 IF TRUE TEST KINDP (FIRST /LIST/) (FIRST /EXP/)
65 IF TRUE OUTPUT FIND (BUTFIRST /LIST/) (BUTFIRST /EXP/)
```

This will mean that to match words on /LIST/ beginning with ANY the words on /EXP/ will have to pass this additional test.

To simplify the writing of KINDP, it is useful to have a procedure FINDPAREN /EXPRESSION/ which outputs the contents of a set of parentheses, if any occur in an expression:

```
TO FINDPAREN /EXPRESSION/
10 TEST IS /EXPRESSION/ /EMPTY/ (If /EXPRESSION/ contains no
20 IF TRUE OUTPUT /EMPTY/        parens, eventually we will get
                                 /EMPTY/ as the answer)
30 TEST IS FIRST /EXPRESSION/ "("
40 IF TRUE OUTPUT BUTFIRST LEFT ")"  (If the first character is
   /EXPRESSION/                      "(", we output everything
                                     between it and the rightmost
                                     ")")
50 OUTPUT FINDPAREN BUTFIRST /EXPRESSION/ (If the first character
                                          is not ")" we try again
END                                       with BUTFIRST /EXPRESSION/)
```

```
←PRINT FINDPAREN "ANY(NUMBER)"
NUMBER
←PRINT FINDPAREN "ABCDEFG"

←
```

Now, KINDP is easy:

```
TO KINDP /ANY.../ /W/
10 MAKE "TYPE" FINDPAREN /ANY.../  (If parentheses in /ANY.../
20 TEST IS /TYPE/ "NUMBER"         contain "NUMBER", output
30 IF TRUE OUTPUT NUMBERP /W/      NUMBERP OF /W/)
40 TEST IS /TYPE/ "NOTNUMBER"      (If parens in /ANY.../ contain
50 IF TRUE OUTPUT NOT NUMBERP /W/  "NOTNUMBER", output NOT OF
60 OUTPUT FALSE                    NUMBERP OF /W/)
END
```

The LOGO predicate NUMBERP /W/ outputs "TRUE" if /W/ is a number and "FALSE" if /W/ is not a number. The procedure NOT used in line 50 outputs "FALSE" if the input is the word "TRUE" and "TRUE" if the input is the word "FALSE". Thus, if /TYPE/ is "NOTNUMBER" KINDP will output "FALSE" or "TRUE" as /W/ is or is not a number.

```
TO NOT /INPUT/
10 TEST IS /INPUT/ "TRUE"
20 IF TRUE OUTPUT "FALSE"
30 OUTPUT "TRUE"
END
```

By using KINDP in its present form, we can restrict our substitutions to cases for which they "make sense". (By further extending KINDP we can be even more discriminating in making substitutions and thereby include more such cases.) Let us, first, change AGEREPLACE to take advantage of our new power. We can do this by changing the words ANYONE'S and ANYNUMBER to ANYONE'S(NOTNUMBER) and ANY(NUMBER), respectively, everywhere they appear.

If we now use AGEREPLACE with the expression that gave us trouble before, the expression emerges unchanged:

```
←PRINT AGEREPLACE "SUM OF JOHN'S AGE AND SUM OF 2 AND 2"
SUM OF JOHN'S AGE AND SUM OF 2 AND 2
```

The new version of AGEREPLACE will correctly translate any age-expression that the previous version did. The only effect of our change is to rule out some situations where the earlier version would have done silly things.

We have written AGEREPLACE to convert many kinds of prefix expressions involving age to more colloquial forms. Let us extend our capabilities to a number of different "story problem" situations by writing some more procedures in a similar vein.

Two kinds of story problems which are very similar to age problems are ones involving wins or losses and distance. Thus, twice Jack's balance after winning three dollars and losing two dollars equals 15 dollars, is one sort of expression which occurs in win-and-lose type problems. The other kind of story problem involves distance -- four miles farther than Augusta is 18 miles from here is an expression of that type. We can easily write procedures similar to AGEREPLACE to arrive at these two new story forms from equivalent prefix formulations.

```
TO WINREPLACE /EXP/
10 MAKE "EXP" REPLACE "SUM OF ANYONE'S(NOTNUMBER) WINNINGS AND
   ANY(NUMBER)" "ANYONE'S(NOTNUMBER) WINNINGS AFTER WINNING
   ANY(NUMBER) DOLLARS" /EXP/
20 MAKE "EXP" REPLACE "DIFFERENCE OF ANYONE'S(NOTNUMBER) WINNINGS
   AND ANY(NUMBER)" "ANYONE'S(NOTNUMBER) WINNINGS AFTER LOSING
   ANY(NUMBER) DOLLARS" /EXP/
30 OUTPUT /EXP/
END
```

```
←PRINT WINREPLACE "DIFFERENCE OF JANE'S WINNINGS AND 1000"
JANE'S WINNINGS AFTER LOSING 1000 DOLLARS
```

```
TO DISTANCEREPLACE /EXP/
10 MAKE "EXP" REPLACE "SUM OF DISTANCE TO ANYPLACE(NOTNUMBER)
   AND ANY(NUMBER)" "ANY(NUMBER) MILES FARTHER THAN
   ANYPLACE(NOTNUMBER)" /EXP/
20 MAKE "EXP" REPLACE "DIFFERENCE OF DISTANCE TO ANYPLACE(NOT
   NUMBER) AND ANY(NUMBER)" "ANY(NUMBER) MILES CLOSER THAN
   ANYPLACE(NOTNUMBER)" /EXP/
30 OUTPUT /EXP/
END
```

```
←PRINT DISTANCEREPLACE "SUM OF DISTANCE TO ST.IVES AND 7"
7 MILES FARTHER THAN ST.IVES
```

Now our repertoire is increasing.

We can also begin to extend our translation facility in another way. We can replace certain phrases by idioms, the ones ordinarily found in story problems. For example, the expression "PRODUCT OF 2 AND" often is said more informally as TWICE. Let us therefore write a procedure IDIOM in which we start incorporating many sorts of standard substitutions.

```
TO IDIOM /EXP/
10 MAKE "EXP" REPLACE "PRODUCT OF 2 AND" "TWICE" /EXP/
20 MAKE "EXP" REPLACE "PRODUCT OF 3 AND" "TRIPLE" /EXP/
30 MAKE "EXP" REPLACE "QUOTIENT OF ANY(NUMBER) AND 2" "HALF OF
   ANY(NUMBER)" /EXP/
END
```

And we can add to this procedure as we find more such examples.

We can, along the lines indicated in this section, translate prefix expressions into several of the standard expressions which so frequently occur in story problems. Our next step is to embed this work in a comprehensive set of procedures for generating entire story problems from infix equations.

3.2 Simple Equations and Simple Story Problems

To set up appropriate equations, we should consider which forms correspond most naturally to the sorts of story problems treated thus far. The basic translation "atom" is the form something + number. Thus, we start by hiding values "under" certain letters.

```
TO GIVE-VALUES /LIST/
10 TEST EMPTYP /LIST/
20 IF TRUE STOP
30 MAKE FIRST OF /LIST/ RANDOM
40 GIVE-VALUES BUTFIRST OF /LIST/
END
```

```
←GIVE-VALUES "A B"
←PRINT /A/
8
←PRINT /B/
3
```

Then we complicate matters by building up components in a manner we know to be easily translatable:

```
TO SIMPLE-EXP /CHAR/
10 OUTPUT WORD OF WORD OF /CHAR/
    AND FLIP OF "+-"
    AND RANDOM
END
```

FLIP randomly chooses between + and - (or, indeed, between any two things given in its input).

```
TO FLIP /PAIR/
10 TEST GREATERP RANDOM 4
20 IF TRUE OUTPUT FIRST OF /PAIR/
30 OUTPUT LAST OF /PAIR/
END
```

```
←PRINT SIMPLE-EXP "A"
A+4
←PRINT SIMPLE-EXP "A"
A-0
```

Note that we can no longer simply place slashes around each expression to find its value. Yet if we are to create equations, we must know the values of each component expression. We can "evaluate" such expressions in a general manner: First we replace each variable (name) by its value (thing), thereby obtaining a numerical infix expression. We can translate this infix expression to prefix form using PREFIXALL, and finally, using the procedure PERFORM, actually "perform" the prefix operations.

```
TO EVALUATE /EXPRESSION/
10 OUTPUT PERFORM OF PREFIXALL OF NUMERIZE OF /EXPRESSION/
END

TO NUMERIZE /EXPRESSION/
10 TEST EMPTYP OF /EXPRESSION/
20 IF TRUE OUTPUT /EMPTY/
30 TEST EMPTYP OF THING OF        (Does the first element have an
   (/FIRST OF /EXPRESSION/)        /EMPTY/ THING?)
40 IF TRUE OUTPUT WORD OF         (If so, go on to the rest of
   (FIRST OF /EXPRESSION/)         the expression)
   AND (NUMERIZE OF BUTFIRST
   OF /EXPRESSION/)
50 OUTPUT WORD OF (THING OF       (Otherwise, replace it with its
   FIRST OF /EXPRESSION/) AND      thing and repeat the process on
   (NUMERIZE OF BUTFIRST OF        the rest of the expression)
   /EXPRESSION/)
END
```

```
←PRINT NUMERIZE OF "A+4"
8+4
←PRINT PREFIXALL OF "8+4"
SUM 8 4
←PRINT EVALUATE OF "A+4"
12
←PRINT EVALUATE OF "X+3"
SUM OF "" AND "3"?               (Error message because /X/ was
INPUTS MUST BE NUMBERS            /EMPTY/, not a number)
I WAS AT LINE 10 IN EVALUATE
←
```

We can use EVALUATE on very much more complicated inputs:

```
←PRINT EVALUATE "(A+4)*(A-3)+A*A*B"
252
```

This capability will be useful later. In any case, we are now ready to create very simple equations:

```
TO SIMPLE-EQ /CHARACTER/
10 GIVE-VALUES /CHARACTER/
20 MAKE "LEFT HAND SIDE"
    SIMPLE-EXP /CHARACTER/
30 OUTPUT WORD OF WORD OF
    /LEFT HAND SIDE/
    "="
    EVALUATE /LEFT HAND SIDE/
END
```

```
←PRINT SIMPLE-EQ "R"
R-3=3
←PRINT /R/
6
```

Thus, we have both created an equation and retained its solution. The next step is to replace the simple one-character variable we are using to hide the solutions by the kinds of names we find in story problems. Once this is done we can use the results of the previous section, AGEREPLACE, for example, to complete the translation process. Let us, for the moment, restrict ourselves to the same three types of problems we discussed then -- age, distance, and winnings type problems. We want to give new "meaningful" names to the variables while keeping track of the original correspondences. This is easily done with a MAKE command by making the new "meaningful" name the thing of the original variable.

```
←TO AGE /LETTER/
>10 MAKE /LETTER/ SENTENCE OF WORD OF
    RND-CHOOSE OF /NAMES/ AND "'S" AND "AGE"
>END
```

```
←MAKE "NAMES" "JOHN HILDA FRED SALLY JAMES IRVING"
```

In each specific case we want to include a unit for each number - either years, dollars, or miles for the three cases studied thus far. We can easily refer to units by giving the names AGE, DISTANCE, WIN, the respective THINGs YEARS, MILES, DOLLARS. Let's now write a procedure to turn an infix expression (or equation) into a meaningful story expression (or equation).

```
←TO INSERT-MEANINGS /EXPRESSION/ /TYPE/  (/TYPE/ is the problem
                                          type - "AGE" for example)
>10 TEST EMPTYP /EXPRESSION/
>20 IF TRUE OUTPUT /EMPTY/
>30 TEST NUMBERP (FIRST OF /EXPRESSION/) (Is element a number?)
>40 IF TRUE OUTPUT SENTENCE OF SENTENCE OF
     FIRST OF /EXPRESSION/               (If so, output it followed
     THING OF /TYPE/                      by its proper unit and
     INSERT-MEANINGS (BUTFIRST            repeat process on rest
     /EXPRESSION/) /TYPE/                 of expression)
>50 TEST EMPTYP OF (THING OF             (Does element have an
     FIRST OF /EXPRESSION/                empty thing?)
>60 IF TRUE OUTPUT SENTENCE OF           (If so, output it and
     FIRST OF /EXPRESSION/                repeat process on rest
     INSERT-MEANINGS OF (BUTFIRST         of expression)
     /EXPRESSION/) AND /TYPE/
>70 OUTPUT SENTENCE OF                   (If not, output its
     THING OF FIRST OF /EXPRESSION/       thing and repeat process
     INSERT-MEANINGS OF (BUTFIRST         on rest of expression)
     /EXPRESSION/) AND /TYPE/
>END
```

We now have all the pieces for a first attempt at creating story problems. Let's try them out, one by one.

```
←SIMPLE-EQ "R"
R+4=12
```

```
←PRINT /R/
8
←AGE "R"
←PRINT /R/
HILDA'S AGE
←PRINT INSERT-MEANINGS "R+4=12" "AGE"
HILDA'S AGE + 4 YEARS = 1 YEARS 2 YEARS
```

This last printout is not yet right - we wanted:
HILDA'S AGE + 4 YEARS = 12 YEARS.
The reason for the discrepancy is readily apparent. On the left-hand side, each character in the original expression stands alone as a complete element -- our variables are single letters and the number is produced by RANDOM, and is thus a single digit. This is not so on the right-hand side, however. Here the element is a two-digit number. To resolve this difficulty we simply add a test to INSERT-MEANINGS:

```
23 TEST NUMBERP /EXPRESSION/     (Is entire expression a number?)
26 IF TRUE OUTPUT SENTENCE OF    (If so, output it followed by
    /EXPRESSION/ AND THING OF     its proper unit)
    /TYPE/
```

```
←PRINT INSERT-MEANINGS "R+4=12" "AGE"
HILDA'S AGE + 4 YEARS = 12 YEARS
```

This can be put in somewhat more natural story form using PREFIXALL as follows:

```
←PRINT PREFIXALL "HILDA'S AGE + 4 YEARS = 12 YEARS"
SUM OF HILDA'S AGE AND 4 YEARS = 12 YEARS
```

And this last text can be made even smoother by AGEREPLACE:

```
←PRINT AGEREPLACE "SUM OF HILDA'S AGE AND 4 YEARS = 12 YEARS"
HILDA'S AGE 4 YEARS FROM NOW = 12 YEARS
```

Now let us put the pieces together so that we can present a meaningful sequence automatically:

```
TO STORY-QUIZ1
10 MAKE "VARIABLE" RND-CHOOSE "ABCDEXYZ"
20 MAKE "EQUATION" SIMPLE-EQ /VARIABLE/
30 MAKE "ANSWER" THING OF /VARIABLE/
40 AGE /VARIABLE/
50 PRINT AGEREPLACE OF PREFIXALL OF
     INSERT-MEANINGS OF /EQUATION/ AND "AGE"
60 PRINT SENTENCE OF SENTENCE OF
     "WHAT IS"
     THING OF /VARIABLE/
     "?"
70 TEST IS "ANSWER" REQUEST
80 IF TRUE PRINT "EXCELLENT!"
90 IF TRUE STOP
100 PRINT SENTENCE OF SENTENCE OF SENTENCE OF SENTENCE OF
     "WHAT WOULD THE EQUATION BE IF WE SUBSTITUTE"
     /VARIABLE/
     "FOR"
     THING OF /VARIABLE/
     "?"
110 TEST IS /EQUATION/ REQUEST
120 IF TRUE PRINT "YOU HAVE THE RIGHT EQUATION.  YOU MUST
     HAVE SOLVED IT INCORRECTLY."
130 IF FALSE PRINT SENTENCE OF "WRONG.  THE CORRECT
     EQUATION IS" /EQUATION/
END
```

Here is an illustration of the use of this program.

```
←STORY-QUIZ1
JAMES'S AGE 3 YEARS FROM NOW = 8 YEARS
WHAT IS JAMES'S AGE ?
:4
WHAT WOULD THE EQUATION BE IF WE SUBSTITUTE E FOR
JAMES'S AGE ?
:E+3=8
YOU HAVE THE RIGHT EQUATION.  YOU MUST HAVE SOLVED
IT INCORRECTLY.
←
```

STORY-QUIZ1 is the prototype for a series of increasingly more complex story problem generators and testers. It is interesting to note that this general method of preparing story problems does not require a knowledge of how to solve such problems. In fact, it is quite easy to make up problems which are extremely difficult to solve, or, even impossible to solve (for example, one equation in two unknowns). Clearly the problems we can create so far, using STORY-QUIZ1, are very easy to solve, even if only by trial and error. It is useful to use the general form of STORY-QUIZ1 as the context within which to present more and more difficult problems. At each stage, though, we must be careful to insure that the obfuscation procedure involved in generating our story problems can, in fact, be reversed to give rise to a solution procedure.

Before going on to more complicated equation forms, however, there are some simple ways in which we can improve STORY-QUIZ1. The story problem we generated with it was
JAMES'S AGE 3 YEARS FROM NOW = 8 YEARS.
This would be more natural if we replaced the "=" by EQUALS and the numbers by equivalent words. The first of these is easily done in INSERT-MEANINGS, by MAKEing "=" "EQUALS". INSERT-MEANINGS will then find the thing of "=" non-empty and will substitute "EQUALS" for "=". Writing numbers out in English form is better done by a procedure:

```
TO ENGLISH-NUM /EXPRESSION/
10 TEST EMPTYP /EXPRESSION/
20 IF TRUE OUTPUT /EMPTY/
30 TEST NUMBERP FIRST /EXPRESSION/
40 IF FALSE OUTPUT SENTENCE OF
    FIRST /EXPRESSION/
    ENGLISH-NUM BUTFIRST /EXPRESSION/
50 MAKE "NAME"
    NTH
      FIRST /EXPRESSION/
      "ZERO ONE TWO THREE FOUR
      FIVE SIX SEVEN EIGHT NINE"
60 OUTPUT SENTENCE OF
    /NAME/
    ENGLISH-NUM (BUTFIRST /EXPRESSION/)
END
```

The subprocedure NTH simply picks out the nth element on a list. The numbering of elements on the list is taken to start at 0.

```
TO NTH /NUMBER/ /LIST/
10 TEST IS /NUMBER/ 0
20 IF TRUE OUTPUT FIRST /LIST/
30 OUTPUT NTH
    DIFF /NUMBER/ 1
    BUTFIRST /LIST/
END
```

```
←PRINT ENGLISH-NUM "4 SCORE AND 7 YEARS AGO"
FOUR SCORE AND SEVEN YEARS AGO
```

Now that it is written, we must decide at what point in the sequence ENGLISH-NUM should be applied. Since numbers in word form are not recognized as numbers by our REPLACE procedure, and cannot be used by our EVALUATE, it is best used as late as possible. Thus, we apply ENGLISH-NUM to our story problem just before it is printed out. We do this by changing Line 50 in STORY-QUIZ1

```
50 PRINT AGEREPLACE OF PREFIXALL OF INSERT-MEANINGS OF
    /EQUATION/ AND "AGE"
```

by

```
50 PRINT ENGLISH-NUM OF AGEREPLACE OF PREFIXALL OF INSERT-MEANINGS
    OF /EQUATION/ AND "AGE"
```

We have made our conversation a little more elegant:

```
←STORY-QUIZ1
IRVING'S AGE FOUR YEARS AGO EQUALS TWO YEARS
WHAT IS IRVING'S AGE ?
*6
EXCELLENT!
```

We now write procedures for the other two types of story problems, using AGE as a model.

```
TO WIN /LETTER/
10 MAKE /LETTER/ SENTENCE OF
    RND-CHOOSE /NAMES/
    "'S WINNINGS"
END
```

```
TO DISTANCE /LETTER/
10 MAKE /LETTER/ SENTENCE OF
    "DISTANCE TO" AND
    RND-CHOOSE /PLACE NAMES/
END
```

```
MAKE "PLACE NAMES" "TASHKENT DACCAR AULIS
    LHASA CHICAGO DOGPATCH VEGA"
```

Given that we have three types of story problem, let us replace the problem-posing algorithm given in Lines 40, 50, 60 of STORY-QUIZ1 by a general procedure MAKE-STORY which uses the <u>type</u> of problem as input. MAKE-STORY is easily followed if one notes the simple correspondence between its lines 10, 20, 30 and lines 40, 50, 60 of STORY-QUIZ1.

```
TO MAKE-STORY /TYPE/ /EQUATION/ /VARIABLE/
10 DO SENTENCE OF /TYPE/ /VARIABLE/
20 PRINT ENGLISH-NUM OF PERFORM SENTENCE OF
    WORD OF /TYPE/ "REPLACE"
    PREFIXALL INSERT-MEANINGS /EQUATION/ /TYPE/
30 PRINT SENTENCE OF SENTENCE OF
    "WHAT IS"
    THING OF /VARIABLE/
    "?"
END
```

Thus, for example:

```
←MAKE-STORY "DISTANCE" "B-7=6" "B"
SEVEN MILES CLOSER THAN VEGA EQUALS SIX MILES
WHAT IS DISTANCE TO VEGA ?
←
```

And we can incorporate MAKE-STORY in a procedure which makes up equivalent problems of the various types given to it in a list.

```
TO MAKE-EQUIV-STORIES /TYPE LIST/ /EQUATION/ /VARIABLE/
10 TEST EMPTYP /TYPE LIST/
20 IF TRUE STOP
30 MAKE-STORY OF
    FIRST /TYPE LIST/
    /EQUATION/
    /VARIABLE/
40 MAKE-EQUIV-STORIES OF
    /BUTFIRST /TYPE LIST/
    /EQUATION/
    /VARIABLE/
END
```

If we want to generate quiz problems pointing up various equivalent forms, we can replace lines 40, 50, 60 in STORY-QUIZ1 by the two lines:

```
40 PRINT "THE FOLLOWING PROBLEMS ALL HAVE THE SAME NUMERICAL
    ANSWER"
50 MAKE-EQUIV-STORIES "AGE DISTANCE WIN" /EQUATION/
    /VARIABLE/
```

We must also change line 100 to eliminate reference to a specific meaning since the same variable now has several interpretations.

```
100 PRINT SENTENCE OF SENTENCE OF "WHAT EQUATION CORRESPONDS
    TO THE ABOVE STORY-PROBLEMS, IF WE USE THE VARIABLE"
    /VARIABLE/
    "TO STAND FOR THE UNKNOWN QUANTITY IN EACH?"
```

```
←STORY-QUIZ1
THE FOLLOWING PROBLEMS ALL HAVE THE SAME NUMERICAL ANSWER
IRVING'S AGE THREE YEARS FROM NOW EQUALS 12 YEARS
WHAT IS IRVING'S AGE ?
THREE MILES FARTHER THAN AULIS EQUALS 12 MILES
WHAT IS DISTANCE TO AULIS ?
HILDA'S WINNINGS AFTER WINNING THREE DOLLARS EQUALS 12 DOLLARS
WHAT IS HILDA'S WINNINGS ?
*4
WHAT EQUATION CORRESPONDS TO THE ABOVE STORY-PROBLEMS, IF WE
USE THE VARIABLE A TO STAND FOR THE UNKNOWN QUANTITY IN EACH ?
*A-3=12
WRONG.  THE CORRECT EQUATION IS A+3=12
←
```

3.3 Rate Problems and Systems of Equations

We could continue our story-problem-writing by directly extending the results of the preceding section. By writing procedures similar to SIMPLE-EQ, to generate a variety of more complex equations, the scope of STORY-QUIZ1 could be greatly extended. Such a development is outlined, in some detail, in the problem set following. We choose, instead, in this section to lay the groundwork for handling a very different kind of story problem - the rate problems.

A typical problem of this form is:

1) If three cats and four houses cost eighteen dollars and
2) two cats and three houses cost thirteen dollars,
3) how much do seven cats and two houses cost?

Two aspects of such problems are of particular interest to us. The story problems discussed in the last section were essentially modifications of the prefix form of the equivalent algebraic equations. These rate problems, however, are directly derivable from the infix form. The example above is simply a word-by-word substitution for

```
if  3*/# CATS/ + 4*/# HOUSES/ = 18
and 2*/# CATS/ + 3*/# HOUSES/ = 13
what is 7* /# CATS/ + 2*/# HOUSES/ ?
```

The other unusual feature of this kind of story-problem is that it lends itself to problems involving as many different unknowns as we might wish. We can as easily make up an example with four quantities to be determined as we can a problem with two such quantities. This is because of the simplicity and generality of the translation process.

The great power we can gain by writing procedures to generate general rate problems brings with it some increased responsibility. It was easy to see that the simple equations we generated in the last section, for example /X/+3=4, were capable of solution. After all, we generated solutions in advance by hiding certain numbers as "unknowns". An equation, or system of equations, is guaranteed to admit our solutions if we generate the equations having these given values of the unknowns in mind. That the solutions we have in mind are the only ones is by no means certain, however, and trivial counter-examples can be shown. For example, if we just take the first of the two equations given above

three cats and four houses cost eighteen dollars

and pose this alone as a separate problem, asking the cost per house and per cat, there are any number of solutions, although many are not meaningful because of external, semantic constraints (e.g., the solution: 1 cat costs -3 dollars, and 1 house costs 6.75 dollars).

For the moment, we will satisfy ourselves with the general observation that if there are n unknowns, at least n equations are required to specify them uniquely (and, of course, this is a sufficient condition if the equations are independent). We will deal with the question of when solutions are unique in some problems. There we will explore the criteria for independence of equations and write procedures which can be used to generate story-problems of guaranteed validity.

The procedure structure required to generate rate problems can be very similar to that required for the earlier type of story-problem. We first choose the list of numbers which we hide under variable names. The procedure we used earlier --

GIVE-VALUES /LIST/ can still be used. The second step is to generate an equation which uses these variable names. Then we need only attach meanings to each of the variables and other symbols. The rest is already done. To generate equations of the form desired for rate problems we first write LINEAR-EXP /LIST/ to create an expression which is the sum of the terms on /LIST/, each term being multiplied by a constant coefficient chosen at random.

We want to create more than one quation here; in fact, we need COUNT /LIST/ equations. MAKE-EQS, gives values to the variables on its input list, then uses MAKE-EQ to build up a list of equations.

```
TO LINEAR-EXP /LIST/
10 TEST EMPTYP /LIST/
20 IF TRUE OUTPUT RANDOM
30 OUTPUT WORD WORD WORD WORD
    RANDOM
    "*"
    FIRST /LIST/
    "+"
    LINEAR-EXP (BUTFIRST /LIST/)
END
```

```
TO MAKE-EQS /LIST/
10 GIVE-VALUES /LIST/
20 OUTPUT MAKE-EQ /LIST/ COUNT /LIST/
END
```

```
TO MAKE-EQ /LIST/ /N/
10 TEST IS /N/ 0
20 IF TRUE OUTPUT /EMPTY/
30 MAKE "EXP" LINEAR-EXP /LIST/
40 OUTPUT SENTENCE WORD WORD
    /EXP/
    "="
    EVALUATE /EXP/
    MAKE-EQ /LIST/ (DIFF /N/ 1)
END
```

These procedures produce a sentence each word of which is an equation. This is an unattractive form for displaying the equations. We write PP /SENTENCE/ to print the equations in the usual way, starting each on a new line.

```
TO PP /SENTENCE/
10 TEST EMPTYP /SENTENCE/
20 IF TRUE STOP
30 PRINT FIRST /SENTENCE/
40 PP BUTFIRST /SENTENCE/
END
```

```
←PP MAKE-EQS "A B C D F G"
5*A+7*B+3*C+8*D+5*F+2*G+1=168
7*A+2*B+0*C+4*D+4*F+1*G+3=127
2*A+5*B+0*C+9*D+9*F+3*G+3=162
0*A+4*B+0*C+6*D+9*F+4*G+8=125
6*A+7*B+3*C+5*D+5*F+7*G+2=169
8*A+8*B+5*C+0*D+0*F+2*G+1=113
```

We can easily look to find what numbers are hidden under the various names.

```
←LIST ALL NAMES

/EXP/ IS "8*A+8*B+5*C+0*D+0*F+2*G+1"
/A/ IS "9"
/B/ IS "3"
/C/ IS "2"
/D/ IS "8"
/F/ IS "5"
/G/ IS "3"
```

We can now generate systems of equations; the next step is to translate them into the "rate" type of story problem we discussed at the beginning of this section. To keep our procedures general, we specify the exact kind of rate problem to be used as input. But we will keep three specific examples in mind throughout,

a) "cost" problems

If ___ ___s and ___ ___s and ... cost ___ dollars. . . .
How many dollars do ___ ___s cost?

b) "building" problems

If ___ ___s and ___ ___s and ... build ___ houses. . . .
How many houses do ___ ___s build?

c) "love" problems

If ___ ___s and ___ ___s and ... love grinches. . . .
How many grinches do ___ ___s love?

These examples are very much alike. They are derived from the equivalent algebraic equations by replacing variable letters with "meaningful" names, "+"s with "ands" and the equal sign with a verb appropriate to the specific context being used. Also the righthand number is followed by a unit which again depends on the context.

We first give a "meaningful" name to each unknown. To avoid ambiguities, we insist that each unknown be given a different name. Thus, the procedure ATTACH, both assigns a name at random to each variable, then repeats the process with that name DELETEd from the list of possibilities.

```
TO ATTACH /VARIABLE LIST/ /NAME LIST/
10 TEST EMPTYP /VARIABLE LIST/
20 IF TRUE STOP
30 MAKE FIRST /VARIABLE LIST/
     RND-CHOOSE /NAME LIST/
40 MAKE "NAME LIST"
     DELETE (THING OF FIRST /VARIABLE LIST/ /NAME LIST/)
50 ATTACH (BUTFIRST /VARIABLE LIST/) /NAME LIST/)
END
```

The deletion subprocedure is:

```
TO DELETE /ELEMENT/ /LIST/
10 TEST EMPTYP /LIST/
20 IF TRUE OUTPUT /EMPTY/
30 TEST IS /ELEMENT/ (FIRST /LIST/)
40 IF TRUE OUTPUT DELETE BUTFIRST /LIST/
50 OUTPUT SENTENCE
       FIRST /LIST/
       DELETE /ELEMENT/ (BUTFIRST /LIST/)
END

←PRINT DELETE "*"  "A*B+3*5"
A B + 3 5
←ATTACH "A B C"  "D E F G"
←PRINT /A/
E
←PRINT /B/
G
←PRINT /C/
D
```

Now, just as with the prefix-type story problems in the section preceding, let us put all the words associated with each of our three problem types on a list. The name of the list is taken to be the problem type, so the list is easily referred to. The words on each list must include the verb which is to replace the equal sign, the unit, and the names we wish to use for variables. In order to keep the role of each word clear we adapt the convention that the verb is first, the unit second, and the names follow. This means, for example that, to obtain the names for assigning variables, ATTACH will use BUTFIRST of BUTFIRST of the list (thereby ignoring the first two things on it -- the verb and unit).

```
←MAKE "COSTS"
    "COST DOLLARS DIAMONDS EMERALDS CATS DOGS
     MOVIES PEANUTS ROCKS TELEPHONES BOOKS FROGS"
←MAKE "BUILDING"
    "BUILD HOUSES MONKEYS GIRAFFES LEOPARDS ELEPHANTS
     EMUS PUMAS WORMS FROGS PLATYPUSSES TERNS"
←MAKE "LOVE"
    "LOVE GRINCHES MEN WOMEN CHILDREN DOGS
     CATS WORMS FROGS MICE GIRLS BOYS"
```

The next step is to put these "meanings" into the systems of equations we derived earlier. We cannot, however, use the procedure INSERT-MEANINGS, that was written in the last section to do this. Our requirements now are much simpler. We have no unit to place after each number, for example. We just want to replace each variable letter by its THING (assigned by ATTACH), each + by "AND" and the "=" with a suitable verb. The only other objects in our equations are numbers, which we leave unchanged and "*" which we want omitted. We will give the correspondences of the "+" and "-" in the procedure which uses our new procedure, INSERT-TYPE. Thus,

```
TO INSERT-TYPE /EXP/
10 TEST EITHER
      EMPTYP /EXP/
      NUMBERP /EXP/
20 IF TRUE OUTPUT /EXP/
30 TEST NUMBERP FIRST /EXP/
40 IF TRUE OUTPUT SENTENCE
      FIRST /EXP/
      INSERT-TYPE BUTFIRST /EXP/
50 OUTPUT SENTENCE
      THING (FIRST /EXP/)
      INSERT-TYPE (BUTFIRST /EXP/)
END
```

Now, finally, we can actually start to change algebraic equations into different kinds of rate problems. We first write a top-level procedure BUILDALL /EQUATIONS/ /TYPE/ /LIST/, which is given a system of equations, the type of problem desired, and a /LIST/ of the variables contained in the equations. BUILDALL sets up all necessary correspondences and then calls BUILD to make the substitutions. We must also modify MAKE-EQ because we do not want the left-hand-side of the equations it produces to end with a constant term. Thus, we change line 30 in MAKE-EQ

```
30 MAKE "EXP" LINEAR-EXP /LIST/
```

to

```
30 MAKE "EXP" BUTLAST BUTLAST LINEAR-EXP /LIST/
```

The two BUTLASTS eliminate both the constant digit and the "+" preceding it.

```
TO BUILDALL /EQUATIONS/ /TYPE/ /LIST/
10 MAKE "+"  "AND"
20 MAKE "*"  /EMPTY/
30 MAKE "="  FIRST (THING OF /TYPE/)
40 ATTACH
      /LIST/
      BUTFIRST BUTFIRST (THING OF /TYPE/)
50 OUTPUT BUILD /EQUATIONS/ /TYPE/
END
```

```
TO BUILD /EQUATIONS/ /TYPE/
10 TEST EMPTYP /EQUATIONS/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT SENTENCE SENTENCE SENTENCE
      INSERT-TYPE (FIRST /EQUATIONS/) (SECOND THING OF /TYPE/)
      ","
      BUILD (BUTFIRST /EQUATIONS/) /TYPE/
END
```

And now, just the few simple procedures we have written in this section permit us to create apparently very complicated structures, such as:

```
←MAKE "EQUATIONS" MAKE-EQS "Q R S T"
←PP /EQUATIONS/
2*Q+0*R+2*S+9*T=40
7*Q+5*R+6*S+6*T=126
6*Q+1*R+8*S+9*T=109
3*Q+2*R+8*S+5*T=101
←PRINT BUILDALL /EQUATIONS/ "LOVE" "Q R S T"
2 CHILDREN AND 0 CATS AND 2 WOMEN AND 9 MICE LOVE 40 GRINCHES , 7
CHILDREN AND 5 CATS AND 6 WOMEN AND 6 MICE LOVE 126 GRINCHES , 6
CHILDREN AND 1 CATS AND 8 WOMEN AND 9 MICE LOVE 109 GRINCHES , 3
CHILDREN AND 2 CATS AND 8 WOMEN AND 5 MICE LOVE 101 GRINCHES ,
```

or:

```
←MAKE "SYSTEM" MAKE-EQS "U V W X Y"
←PP /SYSTEM/
0*U+2*V+9*W+2*X+7*Y=108
6*U+6*V+8*W+6*X+1*Y=150
9*U+9*V+3*W+2*X+8*Y=152
3*U+5*V+7*W+3*X+8*Y=132
2*U+1*V+7*W+2*X+0*Y=75
←PRINT BUILDALL /SYSTEM/ "BUILDING" "U V W X Y"
0 LEOPARDS AND 2 MONKEYS AND 9 ELEPHANTS AND 2 EMUS AND 7 WORMS BUILD
108 HOUSES , 6 LEOPARDS AND 6 MONKEYS AND 8 ELEPHANTS AND 6 EMUS AND 1
WORMS BUILD 150 HOUSES , 9 LEOPARDS AND 9 MONKEYS AND 3 ELEPHANTS AND 2
EMUS AND 8 WORMS BUILD 152 HOUSES , 3 LEOPARDS AND 5 MONKEYS AND 7
ELEPHANTS AND 3 EMUS AND 8 WORMS BUILD 132 HOUSES , 2 LEOPARDS AND 1
MONKEYS AND 7 ELEPHANTS AND 2 EMUS AND 0 WORMS BUILD 75 HOUSES ,
←
```

Our BUILDALL, as it stands cannot correctly deal with single equations because BUILD expects a sentence, containing equations as words. This is easily remedied by adding a line to BUILDALL which changes /EQUATIONS/ to a sentence if it is a word, but otherwise has no effect:

```
5 MAKE "EQUATIONS" SENTENCE OF /EQUATIONS/ AND EMPTY
```

and, with this change,

```
←PRINT BUILDALL "3*X=9" "COSTS" "X"
3 MOVIES COST 9 DOLLARS ,
←
```

We have not yet generated the question part of the rate problem. As we saw from the specific examples we are using, a reasonable form for a question following a translated system of equations is: How many ___s do ___ ___s ___? Of the four blanks in the last part, two depend in a simple way on the type of rate problem. The first one is the unit (DOLLARS, in the case of costs) and the last one the verb (COST, in the case of costs). The third blank is one of the names used in the problem and the second one is a constant multiplier to make the form more interesting. We must write a procedure QUESTION, to construct the one missing part of our rate problems, the question:

```
TO QUESTION /TYPE/ /MULTIPLIER/ /UNKNOWN/
10 OUTPUT  S S S S S S
       "HOW MANY"
        SECOND (THING /TYPE/)
       "DO"
       /MULTIPLIER/
       /UNKNOWN/
        FIRST (THING /TYPE/)
       "?"
END
```

Finally, we can write a precedure ONE-TYPE which uses BUILDALL and QUESTION to do the complete translation. The inputs that ONE-TYPE requires are a combination of the inputs of its two sub-procedures, with only one exception. We choose /UNKNOWN/ for QUESTION to be the name corresponding to the first variable on the list of variables.

```
TO ONE-TYPE /EQUATIONS/ /TYPE/ /VARIABLES/ /MULTIPLIER/
10 PRINT BUILDALL /EQUATIONS/ /TYPE/ /VARIABLES/
20 PRINT QUESTION
            /TYPE/
            /MULTIPLIER/
             THING (FIRST OF /VARIABLES/)
END
```

We can create very difficult problems:

```
←MAKE "TRIAL" MAKE-EQS "W X Y Z"
←PRINT /W/
3
←PRINT /X/
9
←PRINT /Y/
8
←PRINT /Z/
2
←PP /TRIAL/
2*W+0*X+2*Y+9*Z=40
7*W+5*X+6*Y+6*Z=126
6*W+1*X+8*Y+9*Z=109
3*W+2*X+8*Y+5*Z=101
←ONE-TYPE /TRIAL/ "COSTS" "W X Y Z" 3
2 CATS AND 0 MOVIES AND 2 EMERALDS AND 9 TELEPHONES COST 40 DOLLARS , 7
CATS AND 5 MOVIES AND 6 EMERALDS AND 6 TELEPHONES COST 126 DOLLARS , 6
CATS AND 1 MOVIES AND 8 EMERALDS AND 9 TELEPHONES COST 109 DOLLARS , 3
CATS AND 2 MOVIES AND 8 EMERALDS AND 5 TELEPHONES COST 101 DOLLARS ,
HOW MANY DOLLARS DO 3 CATS COST ?
←
```

or more reasonable ones:

```
←MAKE "TWO EQUATIONS" MAKE-EQS "Q R"
←PP /TWO EQUATIONS/
1*Q+7*R=19
0*Q+4*R=8
←ONE-TYPE /TWO EQUATIONS/ "LOVE" "Q R" 7
1 MEN AND 7 CHILDREN LOVE 19 GRINCHES , 0 MEN AND 4 CHILDREN LOVE 8
GRINCHES ,
HOW MANY GRINCHES DO 7 MEN LOVE ?
←
```

We now have all tools prepared for posing rate problems. A possible next step is to write a procedure like STORY-QUIZ1 in the previous section which poses algebraic equations, and various equivalent story-problems to the user. This is left as an exercise. Other possible extensions of the work in this section are discussed in the set of problems following.

APPENDIX A.

TREE, A Tree Drawing Program

The basic idea used in TREE is that if an operator is followed by two inputs which it can evaluate immediately, it does so. In producing a tree diagram corresponding to a given prefix expression, actual evaluation is replaced by the writing of the operator symbol below and halfway between the inputs. This new "node" replaces its inputs in our analysis of the tree, and the process is repeated until the tree has narrowed to just one node.

PREPROC is used to print all numerical inputs in the original prefix expression, leaving a space between each. It replaces each numerical input with its position along the page. Commas and parentheses, always superfluous in prefix form, are eliminated. Thus, PREPROC +(+(3,0),-(7,1)) would actually print 3 0 7 1 and output + + 1 3 - 5 7.

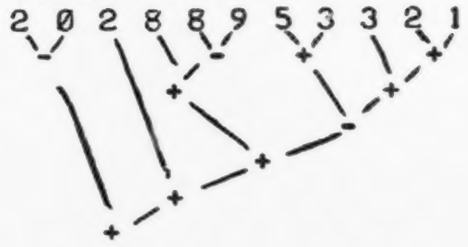
PRINTLINE is then used to print the succeeding rows. PRINTLINE goes through its input, without changing it until it comes to two successive numbers. This circumstance indicates that the operator directly preceding is a current node. This operator is typed halfway between the positions indicated by the two numbers following it (the average of the two numbers). The operator symbol and the two numbers are then replaced by the number corresponding to this new position. PRINTLINE continues until it has gone through the entire expression.

MANYPRINT iteratively calls PRINTLINE until the expression has collapsed to a single node.

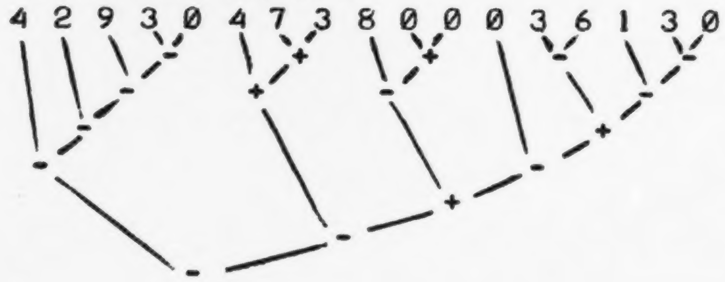
AV outputs the average of its inputs and SPACE spaces /N/ times. RND-PREFIX randomly generates prefix expressions of depth equal to its input. RND-PREFIX uses RND which gives a random number between 1 and /N/, /N/ being less than 10, and FLIP randomly outputs one or the other of its inputs.

Two examples are given. In the illustrations, the nodes are connected by hand-drawn lines for clarity.

```
←MAKE "EXP" RND-PREFIX 6
←PRINT /EXP/
+(-(2,0),+(2,+(+(8,-(8,9)),-(+(5,3),+(3,+(2,1)))))
←TREE /EXP/
```



```
 MAKE "EXPRESSION" RND-PREFIX 7
←PRINT /EXPRESSION/
-(-(4,-(2,-(9,-(3,0)))),-(+(4,+(7,3)),+(-(8,+(0,0)),-(0,+(-(3,6),-(1,-(3
,0)))))))
←TREE /EXPRESSION/
```



```
←



TO TREE /EXP/
10 MANYPRINT PREPROC /EXP/ -1
END

TO PRINTLINE /EXP/
10 TEST EMPTYP /EXP/
20 IF TRUE PRINT /EMPTY/
30 IF TRUE OUTPUT /EMPTY/
40 TEST BOTH NUMBERP FIRST BUTFIRST /EXP/ NUMBERP FIRST BUTFIRST
   BUTFIRST /EXP/
50 IF TRUE SPACE DIFFERENCE DIFFERENCE AV FIRST BUTFIRST /EXP/ FIRST
   BUTFIRST BUTFIRST /EXP/ /L/ 1
55 IF TRUE TYPE FIRST /EXP/
60 IF TRUE MAKE "L" AV FIRST BUTFIRST /EXP/ FIRST BUTFIRST BUTFIRST
   /EXP/
70 IF TRUE OUTPUT SENTENCE /L/ PRINTLINE BUTFIRST BUTFIRST BUTFIRST
   /EXP/
80 OUTPUT SENTENCE FIRST /EXP/ PRINTLINE BUTFIRST /EXP/
END
```

```
TO SPACE /N/
10 TEST ZEROP /N/
20 IF TRUE STOP
30 TYPE /BLANK/
40 SPACE DIFFERENCE /N/ 1
END

TO AV /A/ /B/
10 OUTPUT QUOTIENT SUM /A/ /B/ 2
END

TO MANYPRINT /EXP/
10 MAKE "L" 0
20 TEST IS COUNT /EXP/ 1
50 IF TRUE STOP
60 MANYPRINT PRINTLINE /EXP/
END

TO PREPROC /EXP/ /N/
10 TEST EMPTYP /EXP/
15 IF TRUE PRINT /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 TEST EITHER EITHER IS FIRST /EXP/ "," IS FIRST /EXP/ "(" IS FIRST
   /EXP/ ")"
40 IF TRUE OUTPUT PREPROC BUTFIRST /EXP/ /N/
50 TEST NUMBERP FIRST /EXP/
60 IF FALSE OUTPUT SENTENCE FIRST /EXP/ PREPROC BUTFIRST /EXP/ /N/
70 TYPE WORD FIRST /EXP/ /BLANK/
80 MAKE "N" SUM /N/ 2
90 OUTPUT SENTENCE /N/ PREPROC BUTFIRST /EXP/ /N/
END

TO RND-PREFIX /N/
10 TEST ZEROP /N/
20 IF TRUE OUTPUT RANDOM
30 MAKE "M" RND DIFFERENCE /N/ 1
40 OUTPUT WORD WORD WORD WORD WORD FLIP "+" "-" "(" RND-PREFIX /M/ ","
   RND-PREFIX DIFFERENCE /N/ SUM /M/ 1 ")"
END

TO RND /N/
10 OUTPUT QUOTIENT DIFFERENCE PRODUCT RANDOM /N/ 1 9
END

TO FLIP /A/ /B/
10 TEST GREATERP RANDOM 4
20 IF TRUE OUTPUT /A/
30 OUTPUT /B/
END
```

APPENDIX B.

SEPARATE, An "Unpacking" Procedure

SEPARATE, a procedure referred to in the text expands each occurrence of /LIST/ characters appearing in /EXP/ so that the characters stand as separate words. SEPARATE makes use of UNPACK which "unpacks" any single character /SYMBOL/.

```
TO SEPARATE /LIST/ /EXP/
10 TEST EMPTYP /LIST/
20 IF TRUE OUTPUT /EXP/
30 MAKE "EXP"
    UNPACK (FIRST /LIST/) /EXP/
40 MAKE "LIST" BUTFIRST /LIST/
50 OUTPUT SEPARATE /LIST/ /EXP/
END

TO UNPACK /SYMBOL/ /EXP/
10 MAKE "EXP" SENTENCE OF /EXP/ AND /EMPTY/
20 TEST EMPTYP /EXP/
30 IF TRUE OUTPUT /EMPTY/
40 TEST EXISTENCE /SYMBOL/ FIRST /EXP/
50 IF TRUE OUTPUT SENTENCE SENTENCE SENTENCE
    UNPACK /SYMBOL/ (LEFT /SYMBOL/ FIRST /EXP/)
    /SYMBOL/
    RIGHT /SYMBOL/ /EXP/
    UNPACK /SYMBOL/ BUTFIRST /EXP/
60 OUTPUT SENTENCE OF
    FIRST /EXP/
    UNPACK /SYMBOL/ BUTFIRST /EXP/
END

TO EXISTENCE /CHARACTER/ /EXPRESSION/
10 TEST EMPTYP /EXPRESSION/
20 IF TRUE OUTPUT "FALSE"
30 TEST IS /CHARACTER/ FIRST /EXPRESSION/
40 IF TRUE OUTPUT "TRUE"
50 OUTPUT EXISTENCE /CHARACTER/ (BUTFIRST
    /EXPRESSION/)
END
```

For example,

```
←PRINT UNPACK "*" "ABC*DE * F**"
ABC * DE * F * *
```

SEPARATE+, also referred to in the text, is just a special case of UNPACK.

```
TO SEPARATE+ /EXP/
10 OUTPUT UNPACK "+" /EXP/
END
```

APPENDIX C.

RND-CHOOSE, A Random Choice Procedure

```
TO NINES /N/                      (Output a word of /N/ nines)
10 TEST IS /N/ 1
20 IF TRUE OUTPUT "9"
30 OUTPUT WORD NINES (DIFF /N/ 1) 9
END


TO NDIGIT /N/                     (Outputs random /N/ digit number)
10 TEST IS /N/ 1
20 IF TRUE OUTPUT RANDOM
30 OUTPUT WORD OF NDIGIT (DIFF /N/ 1)
    AND RANDOM
END


TO RAND /N/                       (Outputs a random number between
                                   1 and any integer /N/)
10 OUTPUT SUM QUOTIENT DIFFERENCE
    (PRODUCT (NDIGIT COUNT /N/) /N/) 1
    (NINES COUNT /N/) 1
END


TO NTH /N/ /LIST/                 (Gives /N/th element of /LIST/)
10 TEST IS /N/ 1
20 IF TRUE OUTPUT FIRST /LIST/
30 OUTPUT NTH (DIFF /N/ 1) BUTFIRST /LIST/
END


TO RND-CHOOSE /LIST/              (Outputs a random choice from /LIST/)
10 OUTPUT NTH (RAND COUNT /LIST/) /LIST/
END
```

PROBLEMS

Introduction

These problems are intended to supplement LOGO and Story Problems, Teacher's Text. For the most part they represent significant extensions of or substantial alternative approaches to the textual presentation. Thus, they will probably serve to illustrate the depth and richness of the LOGO approach, as well as a source of generating specific problems to be posed to students. The numbering of sections in this problem set parallels that of the main text.

Problems

Sections 2.1 and 2.2

The LOGO commands WORD and SENTENCE are operations requiring two inputs. Thus, we can display LOGO expressions made up of these commands in tree form, just as we did with +, -, *, /.

1. Give the tree diagrams corresponding to

(A) SENTENCE OF SENTENCE OF "A" SENTENCE OF "B" WORD OF WORD OF "C" AND "D" AND "E" AND "F"

(B) SENTENCE OF SENTENCE OF WORD OF "G" AND "H" AND WORD OF "J" AND "K" AND SENTENCE OF WORD OF "L" AND "M" AND "N"

2. It is clear that not every combination of the commands WORD and SENTENCE is permissible. For example, one cannot use WORD OF "A" AND SENTENCE OF "C" AND "D". Give a general rule describing the valid combinations of these two commands. What does this rule say, in terms of valid trees?

3. WORD and SENTENCE, as used in LOGO, are in prefix form. One can, of course, also represent these operations in infix form.

For example, we could write "CHICKEN" SENTENCE "LITTLE". Write some complicated prefix expressions using WORD and SENTENCE then translate them to infix form.

4. Are there any rules of precedence which govern whether WORD or SENTENCE gets performed first, in infix notation? Is there any direction in which one should go for correct translation from infix to prefix?

5. Are the operations WORD and SENTENCE associative? What implications does this have for writing infix expressions containing these operators?

Section 2.4

Any subtraction can be represented as an addition of signed numbers. For example, 2 - 3 is equivalent to 2 + (-3). Any repeated subtraction can, by the same principle, be converted to a repeated sum. For example, 2 - 3 - 4 = 2 + (-3) + (-4). Thus, another way of translating infix subtractions into prefix form is to first convert the infix subtraction into a sum, as above, then use the infix sum translation procedure.

1. Write a procedure which translates a single infix difference into the corresponding sum.

2. Extend the procedure to include the translation of repeated differences.

3. Use the procedure just written for repeated differences with the procedure for translating repeated sums to write a new procedure which translates both sums and differences from infix to prefix form.

4. Two sorts of prefix notation are discussed in the text. One is the LOGO notation, the other uses commas and parentheses. For example, consider the equivalent expressions SUM OF DIFFERENCE OF 1 AND 2 AND 3 and +(-(1,2),3). Write two procedures to translate each form into the other. One check on the translation procedures is to note that the composition of either one with the other should leave any valid input expression unchanged.

5. Given three inputs in a specified order, and one kind of operation requiring two inputs such as SUM, how many different trees can be constructed? What about four inputs?

6. Write a procedure ASSOCP /OPERATOR/ /OP1/ /OP2/ /OP3/ which checks whether /OPERATOR/ is associative or not. ASSOCP can do this by applying /OPERATOR/ in two different ways to the three operands, /OP1/, /OP2/, /OP3/, and comparing the results.

7. How many different sets of inputs to ASSOCP should be tried in order to decide whether or not /OPERATOR/ is associative? Write a procedure TRIAL /OPERATOR/ which keeps on generating three random integers and uses ASSOCP with /OPERATOR/ and the three integers as inputs. Use TRIAL to decide whether the operation which doubles its first operand and adds it to the second is associative.

8. Use TRIAL on the operator which doubles its first input and adds it to twice the second input.

9. BOTH and EITHER are LOGO procedures whose inputs are TRUE or FALSE. Modify TRIAL so that it can test the associativity of these two operators. How many different sets of inputs should be tried to see whether or not they are associative.

10. Consider the operator SUMALL which takes any number of integers as operands and gives their sum. For example, SUMALL OF 2 AND 3 AND 4 AND 5 is 14. Can SUMALL be written in a simple infix notation? Is there any ambiguity in writing expressions containing several SUMALL operators?

11. Write a LOGO procedure which translates SUMALL operators in arithmetic expressions to equivalent expressions which use SUMS instead.

Section 2.5

1. Apply PERFORM to the output of your new SUMALL translation procedure as a further check.

Section 2.6

The procedures used for the infix-prefix test given in the text can be generalized in several ways:

1. Write a procedure TIMING /N/ which uses PREFIX-TEST /N/ to pose questions on the translation of an infix expression containing /N/ operators. TIMING is to output a sentence of two words, the first of which is the number of seconds that the student took to answer the question, the second TRUE or FALSE, depending on the correctness of the response.

2. Using TIMING /N/, write a procedure TESTING /N/ /M/ which administers a succession of /M/ questions, each infix expression submitted containing /N/ operators. At the conclusion of the test, the procedure should give the number of questions answered correctly and incorrectly and the total amount of time required to answer all the questions.

3. Modify TESTING so that the number of operators for each question need not be specified as an input. Rather, TESTING should start by presenting an infix expression containing two operators. After each question, TESTING should increase or decrease the number of operators by 1, depending on whether the student answer is right or wrong. Again, the student score and the time taken should be given at the end of the test.

4. The expression "1+2+3" corresponds to two different prefix expressions: SUM OF 1 AND SUM OF 2 AND 3; SUM OF SUM OF 1 AND 2 AND 3. Alternative translations can be performed because we have not exactly specified, in the infix form, what the operands of each of the two + signs are to be. This ambiguity in prefix form is ordinarily acceptable because the associativity of addition guarantees that the two forms evaluate to the same number. In the infix-prefix translation quiz, however, we need to find out whether the student's answer is written in a prefix form equivalent to the one generated by the computer. The method used in the text, if the computer and student prefix versions are not precisely the same, is simply to compare the *values* of the two expressions. This is not sufficient, since comparison of values would tell us that the two prefix expressions SUM OF 1 AND 2 and DIFFERENCE OF 4 AND 1 are equivalent. We could remove ambiguity by suitable placement of parentheses. The two possibilities for the example above are (1+2)+3 and 1+(2+3). Each one of these is equivalent to just one of the prefix forms given. By this means we can always get rid of such ambiguities in infix notation,permitted by associativity.

(A) Formulate a simple rule for determining when an infix expression is ambiguous, i.e., has more than one translation into prefix.

(B) Given such an ambiguous expression, determine how parentheses should be placed to give each equivalent non-ambiguous infix form.

(C) Write a computer program which automatically performs the algorithm derived in part (B).

(D) Use the program from part (C) as part of a program which, given an infix expression, generates all the equivalent prefix translations.

(E) Modify the infix-prefix quiz so that the validity of the student response is determined by comparison with each of these equivalent prefix forms.

(F) Modify the test still further by comparing an incorrect student response to each of the possible answers, and by counting the number of words which match, determine the "closest" one of the correct answer forms. Have the computer print out this "closest" answer and have it also print those parts of this answer and the student response which differ.

5. In the infix-prefix test, we wrote a random infix expression generator. One can also write, although with slightly greater difficulty, a random *prefix* expression generator.

(A) One way to generate prefix expressions, containing /N/ operators is to write a list of /N/ randomly chosen operators, followed by SUM OF /N/ AND 1 randomly chosen operands. Write a procedure to do this. Clearly not all prefix expressions are of this form. To what general class of trees do the expressions generated in this way correspond?

(B) A way to generate arbitrary prefix expressions is to note that any prefix expression can be itself an operand and, furthermore, that all operands must be valid prefix expressions. Use this fact to write a recursive procedure which randomly generates the class of all prefix expressions. (Assume again that the operators and operands are chosen from given lists.)

6. For the most part, the operators which we have studied have an infinite number of possible operands. SUM, for example, requires two operands, each of which is any number whatever. SENTENCE uses operands, each of which is any combination of characters. It is interesting to contrast these with operators whose inputs must be members of a small set. Two such operators are BOTH and EITHER, in LOGO. Each operator requires two inputs, each of which must be the word "TRUE" or "FALSE". BOTH outputs "TRUF" if both inputs are the word "TRUE". If this condition is not satisfied, BOTH outputs "FALSE". EITHER outputs "TRUE" if either one or both of the inputs is the word "TRUE" and "FALSE" if the two inputs are "FALSE".

(A) Draw the tree corresponding to the prefix expression BOTH EITHER EITHER "TRUE" "FALSE" BOTH "TRUE" "FALSE" "TRUE". What would such a LOGO instruction line output?

(B) Are BOTH and EITHER associative operations?

(C) BOTH and EITHER, as used in LOGO, are in prefix form. One could just as well, for example, use the infix form "TRUE" B "FALSE" or "TRUE" E "FALSE" for the two. Translate the expression given in part (A) to this form.

(D) Write an infix-prefix translator to translate the infix form of BOTH and EITHER, as given in part (C) to the prefix form.

Section 2.7

The infix-prefix translator, as it stands, will give incorrect results when the number of left parentheses in the input is not equal to the number of right parentheses.

1. What sorts of error messages would result from such inputs?

2. Modify the procedure INSIDE to detect unbalanced parentheses and output a suitable error message.

3. The procedure PREFIX, in its final form in the text, translates +, -, *, / from infix to prefix form. Modify PREFIX to enable it to also translate the operations WORD and SENTENCE when these are given in prefix form. (See problems for Section 2.2.)

Section 3.1

1. Write a "random" substitution procedure which replaces each different character in its input by one chosen randomly.

2. Use a version of REPLACE to write a new procedure which converts all numbers to their standard English word equivalents. Thus, "12" results in the output "TWELVE". To simplify the work, restrict the domain of inputs to two-digit numbers.

3. Write a procedure which replaces PRODUCT OF (number) AND (something) by (number) (plural of something), wherever this form occurs. Thus, the expression "ANN HAS PRODUCT OF 2 AND EYE" is to be replaced by "ANN HAS 2 EYES".

Section 3.2

1. (A) Write a procedure HARDER-EQ which generates equations of the form $a(x + b) + c = d(x + e) + f$, where a, b, c, d, e, f are constants provided by HARDER-EQ. One way to do this is by use of a procedure BALANCE which "balances out" its two input expressions by adding a suitable number to one of them.

(B) Modify STORY-QUIZ1 so that it uses HARDER-EQ in place of SIMPLE-EQ to provide the starting equation.

(C) Write a procedure TWO-EQS which generates pairs of equations in two unknowns. To be translatable into the kind of story problem we discussed in this section, the equations could be of the form $a(x + b) + c(y + d) = e$, where a, b, c, d, e are constants provided by TWO-EQS.

(D) Modify STORY-QUIZ1 so that the type of equation is an input. Write a procedure which uses STORY-QUIZ1 to systematically vary the type of equation on which the story problems are based.

2. Extend GIVE-VALUES and EVALUATE to permit the use of equations with fractional solutions. Minor changes will also be required in the other procedures used in STORY-QUIZ.

Section 3.3

1. (A) Write a procedure SEARCH which looks through a system of equations, outputting a list of the variables they contain.

(B) Use SEARCH in BUILDALL to eliminate one of the latter's inputs.

2. Another interesting "infix" type of story problem is the "money" problem. An example is:
Some pennies and some dimes are worth 87 cents. How many of each coin are there?
There are many solutions to such an equation but a unique solution can often be specified by a further constraint. We can insist that there must not be too large a quantity of any coin to permit an exchange for a coin higher in value. With this constraint the equation above has a unique solution -- 8 dimes, 7 pennies.

(A) Write a set of procedures which pose such money problems and check a user's answer.

(B) Under what circumstances is there more than one solution to a given problem, even with the above constraint?

(C) What further constraint (s) can be used to make the solution unique in all cases?

3. Thus far we have not considered at all the solutions of the sets of equations we have been generating. Perhaps the simplest way to solve them, conceptually at least, is by systematically trying out solutions. This is a useful method only because the answers to the equation we have been generating lie within a small set of possible values.

(A) Write a procedure TRYOUT /EQUATIONS/ /VARIABLES/ which tries values of the /VARIABLES/ ranging from 0 to 9 in /EQUATIONS/, outputting successful sets of values. It is useful to note in writing TRYOUT that if we regard the set of values of all variables as an N-digit number then the set of all possible values for the variables is given by all N-digit numbers from 0 0 ... 0 to 99 ... 9.

(B) Assuming that each trial takes 1/100 second, how long on the average do you think it would take to solve 4 equations in 4 unknowns by this method? 10 equations in 10 unknowns?

4. Let us now consider a more systematic way of solving a set of equations. This method will also give us a means of determining whether there is a unique solution. We use the facts that a) multiplying both sides of any equation does not alter the values of variables for which it holds true, and that b) given a set of two equations, their difference can replace either of the two, the new set of two equations holding true for the same values of their variables as did the original pair.

(A) Write a procedure, MULT/EQ/ /N/ which, given an equation /EQ/ of the form ax + by + ... = M multiplies each constant a, b... on the left hand side and the number M on the right by /N/. Write a procedure SUB /EQ1/ /EQ2/ which outputs the difference of /EQ1/ and /EQ2/.

(B) Using the two procedures, written in part (A), write a procedure which gives the solution of a pair of simultaneous equations in two unknowns. It can do this by multiplying each of the equations by the leading constant term of the other, then subtracting the two resulting equations. This gives a simple equation of the form by = c with just the second variable remaining.

(C) Under what circumstances does the procedure in part (B) fail? How many solutions do the two equations have in such cases?

(D) Show that a pair of equations in two unknowns has at least one solution, using (B) as a constructive proof.

(E) Generalize the procedure given in (B) to work for an arbitrary number of equations, by systematically eliminating one variable from all but one of the equations. This can be accomplished by suitable multiplications and subtractions, repeating the process with the set of reduced equations until we are left with one equation in one unknown. The only precaution we must take is to ensure that the leading coefficient of the equation we are using to reduce the others has a non-zero leading coefficient.

(F) Show that if when we try to eliminate a variable, all equations containing it have zero as its coefficient, that the equations have infinitely many solutions.